Path Directed Abstraction and Refinement in SAT-based Design Debugging

Brian Keng
University of Toronto
ECE Department, Toronto, Canada
briank@eecg.toronto.edu

Andreas Veneris
University of Toronto
ECE & CS Department, Toronto, Canada
veneris@eecg.toronto.edu

ABSTRACT

The past decade has seen a disproportionate amount of resources dedicated towards verification as compared to actual design. It is reported that one third of this overhead is due to the resource-intensive task of manual debugging. To relieve this burden, this work introduces the novel concept of path directed debugging within a window-based abstraction/refinement framework. The algorithm divides the error trace into non-overlapping time-windows where each window is analyzed separately. Subsequent windows are replaced with abstracted over-approximations derived from failing paths in the time domain. Using this abstracted model, each solution found is processed through an additional verification step that removes spurious solutions and simultaneously refines the problem. This paper also develops the theory that shows that the proposed approach is complete, a fact that mitigates the incompleteness inherent in past time-window based debugging methods. Experimental results on industrial designs with long error traces show a 55% decrease in peak memory usage resulting in 78% more instances being solved when compared to previous work.

Categories and Subject Descriptors

B.5.2 [**Design Aids**]: Verification; B.6.2 [**Design Aids**]: Verification

General Terms

Algorithms, Verification

Keywords

debug, diagnosis

1. INTRODUCTION

Computer-Aided Design (CAD) tools are continuously improving their scalability and efficiency to mitigate the high cost associated with the design of modern VLSI systems. In the past decade, the effort to verify these systems has increased disproportionately [4], a trend coined as the *verification gap*. This is also confirmed by the 3:1 ratio between the number of verification engineers and designers. To make matters worse, this trend has been projected to increase almost seven-fold by 2015 [8]. The resource-intensive task of *debugging* is a significant component of this gap. Technical road-maps and market studies indicate that once a design fails verification, fixing it can take up to 32% of the total verification effort [4].

Automated design debugging techniques [5, 11] aim to increase debugging efficiency by localizing the root-cause of the error. However, the tremendous growth in modern design size (>5M gates per typical design block) and error trace lengths (tens of thousand of cycles) can limit their applicability. The primary cause for this limitation is their use of *time-frame expansion* [11] to model the problem, where the combinational part of the circuit is replicated for the length of the error trace. As such, industrial design sizes coupled with excessive error trace lengths inevitably lead to memory explosion issues.

Recent research to reduce this massive memory footprint focuses on these two pertinent complexity factors (*i.e.,* design size and trace length). Abstraction and refinement techniques [7, 10] have shown to greatly reduce the effective design size by iteratively refining the abstract design until a minimal set of necessary components are determined. Orthogonally, *time-windowing* techniques [6, 13] use a sliding window of consecutive clock cycles to analyze the problem. In this fashion, they model only a segment of the entire error trace and they use different methods to approximate the remaining non-modeled parts. These methods greatly reduce the problem size but their use of approximations leads to many spurious solutions negating the benefit of localization.

In this work, we present a novel design debugging algorithm utilizing time-windows in an abstraction and refinement Boolean Satisfiability (SAT) based framework. The algorithm is also shown to be complete in contrast to previous approximate time-windowing methods that provide no means of refinement [6]. The process begins by dividing the error trace into non-overlapping time-windows where each window is analyzed separately. Just like other window-based methods, only one time-window is explicitly modeled while subsequent time-frames are over-approximated using the abstraction.

The key novelty to our work is that the proposed abstraction initially consists of paths in the time-frame expanded circuit that directly lead to the observed failure during simulation. As before, during each iteration, solutions to this abstract problem may either be valid or spurious. Therefore, an additional verification step is necessary for each solution. This step works by iteratively propagating solutions forward to successive time-windows. If the solution is not spurious then successive time-windows will be satisfiable. Otherwise, they will return `UNSAT` and additional paths are extracted from the resulting conflict to refine the abstract problem. The net result is a dramatically reduced memory footprint that – as theoretically proven in this paper – mitigates the incompleteness issues of past time-windowing methods.

Experimental results on large industrial designs with long error traces from `OpenCores` [9] and from our industrial partners illustrate the benefits of this work. Across all instances, the proposed approach solves 78% more cases when compared to a previous time-windowing technique. This results in an average 55% reduction in peak-memory while being able to debug traces that are 31% longer on average. The abstraction is also shown to be very efficient in reducing problem size using only an average of 4.3% of the clauses needed compared to explicit modeling of the problem.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC 2012, June 3-7, 2012, San Francisco, California, USA.
Copyright 2012 ACM ACM 978-1-4503-1199-1/12/06 ...$10.00.

The remaining sections of this paper proceed as follows. Section 2 presents background material. Section 3 and Section 4 describe the proposed approach and extensions, respectively. Section 5 presents experimental results and Section 6 concludes this work.

2. PRELIMINARIES

2.1 SAT-based Design Debugging

Automated design debugging techniques aim to find error locations, known as *suspects*, that can explain an observed failure during verification [5] given an error trace (*i.e., counter-example*) and a user-defined number of errors. SAT-based design debugging [11] encodes this problem as a SAT instance where the solutions correspond to the set of all possible suspects for the given number of errors.

This SAT instance is constructed in several steps. First, the combinational part of the circuit (T) is enhanced with an error model for each location in the design, denoted as T_{en}. Each error model has an associated *suspect variable* (e_i) that when active (*i.e.,* $e_i = 1$), disconnects the location's fan-out from its fan-in, allowing it to be free. Next, the enhanced combinational circuit is replicated as a time-frame expanded model for the length of the error trace. Observe, the suspect variables are not replicated since they represent the same location regardless of time-frame. On the unrolled model, the initial state (S^p), vector of inputs (X^i), and vector of *expected* or correct outputs (Y^i) are constrained for each clock cycle i to model the error trace. Finally, the number of active suspect variables are constrained ($\Phi(N)$) to exactly N to denote the search for N simultaneous errors. A *debugging solution* to this instance consists of the set of N active suspect variables which, once found, are blocked in order to find all other solutions. The following equation models a window of an error trace with width w from cycle p to $p + w - 1$.

$$Debug_p^{p+w-1}(N) = S^p \wedge \Phi(N) \wedge \bigwedge_{i=p}^{p+w-1} X^i \wedge Y^i \wedge T_{en}^i \quad (1)$$

Solutions to this equation correspond to suspects that can explain the observed error. Note that the observed failure must occur within cycles p to $p + w - 1$ to generate a mismatch between the erroneous circuit behavior and expected outputs. If this is not the case, then the equation is trivially satisfiable at $N = 0$ indicating that the failure cannot be observed in the modeled window and additional cycles must be added.

2.2 Time Diagnosis

An alternate formulation of a debugging instance, known as *time diagnosis* [12], can be generated by utilizing a variation of Eq. 1. The key difference with the one from Section 2.1 is that suspect variables for a location are shared only within a fixed time-window of clock cycles and not over the entire error trace. In other words, this time-window can model the excitation of a suspect within the given set of clock cycles, but it cannot model it across different time-windows. It has been shown [12] that this trade-off can dramatically reduce the problem complexity in an effort to model real-life bugs with long error traces. Due to its relation to the work presented here, we introduce it in some more detail.

To model a debugging instance for a single time-window Eq. 1 can be used. However, as explained earlier, if the observed error is not within this time-window the instance will be trivially satisfiable. To overcome this issue, one can model the circuit behavior for subsequent time-frames in order for the observed failure to be included. The modeling of subsequent times-frames is shown in the next equation for

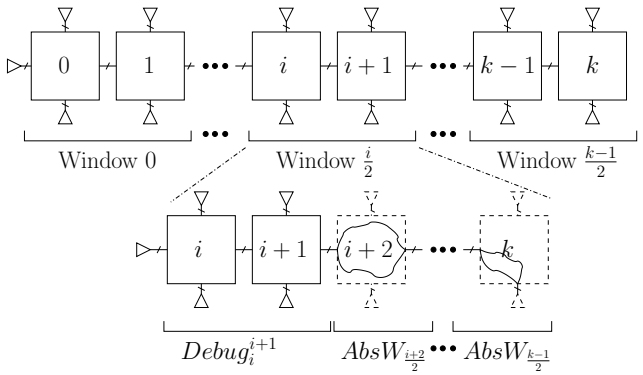


Figure 1: Path Directed Abstraction and Refinement

a *width* of w time-frames starting from cycle $p \cdot w$:

$$W_p = \bigwedge_{i=p\cdot w}^{(p+1)\cdot w-1} X^i \wedge Y^i \wedge T^i \quad (2)$$

Note that if $N = 0$ is set in Eq. 1, it simplifies to a time-frame shifted version of Eq. 2 (with the addition of the initial state constraints S^p) because it eliminates all the additional circuitry that relates to the error models.

By combining Eq. 1 and 2, one can overcome the issue of debugging a time-window that does not contain the observed failure. This formulation is shown with respect to a set of time-windows from p to q with a width of w:

$$TimeDebug_p^q(N) = Debug_{p\cdot w}^{(p+1)\cdot w-1}(N) \wedge \bigwedge_{i=p+1}^{q} W_i \quad (3)$$

For the interested reader, a detailed example of time diagnosis is provided in the supplemental material in Section S1.1.

3. PATH DIRECTED ABSTRACTION AND REFINEMENT

Path directed abstraction and refinement is a methodology built around the debugging model presented in Eq. 3. It also provides a complete technique to iteratively analyze long industrial traces. It begins by dividing an error trace into multiple non-overlapping time-windows. Each window is iteratively analyzed starting from the last time-window in the error trace that contains the observed failure. This "sliding window" of time-frames iteratively moves towards earlier cycles of the trace searching for suspects without the need to explicitly model later windows. To properly constrain the observed failure, each of these subsequent windows are abstracted to a smaller set of constraints. Intuitively, these constraints are modeled with information obtained by structural circuit paths in the time-frame expanded circuit that propagate to the observed failure on the primary output(s).

This key idea is shown in Figure 1 with an error trace containing $k + 1$ time-frames. In this example, the trace is divided up into $\frac{k+1}{w}$ non-overlapping time-windows with $w = 2$. The abstract time diagnosis instance for time-window $\frac{i}{w}$ is shown in greater detail, where the first w time-frames are modeled explicitly with the use of Eq. 1. In the proposed method, the subsequent time-frames (past cycle $i+1$) are not modeled explicitly. Instead, they are abstracted and replaced with information obtained by paths in the time-frame expanded circuit such that the original observed failure can still be modeled. This abstraction over-approximates the subsequent time-frames similar to previous time-windowing techniques [6]. However, the key benefit

is that it provides a compact abstraction combined with an efficient refinement strategy, which is not available in [6], to remove spurious solutions that do not occur in the concrete problem.

These spurious solutions are removed by a refinement step that propagates the associated satisfying assignments forward to each successive explicitly modeled time-window using a separate SAT instance. If all successive windows yield a SAT result, then the solution is valid. Otherwise, a conflict is generated indicating that the current abstraction needs to be refined. The refinement step adds back all clauses involved in the conflict, intuitively adding more paths to the original abstraction. This process is repeated until the refined abstract problem is unsatisfiable. In this case, the current time-window is finished with complete results and the algorithm proceeds to analyze the next one.

The following subsections describe the formulation, theoretical results and pseudo-code for the path-directed abstraction and refinement algorithm in much greater detail. Due to space requirements, formal proofs of the lemmas and theorems as well as a descriptive example can be found in the supplemental material section of this paper.

3.1 Path-based Abstraction

The path-based abstraction approximates Eq. 3 by replacing consecutive windows of time-frames (W_i) with an abstract version, denoted by $AbsW_i$. This is initially formulated by adding paths in the time-frame expanded circuit that are directly involved in the observed failure. Practically, this is a set of clauses extracted from the SAT-solver conflict graph generated while modeling the observed failure. The rest of this subsection describes this process in greater detail.

Consider the first abstract time-window occurring at the end of the trace ($AbsW_q$) for cycles $q \cdot w$ to $(q+1) \cdot w - 1$. This abstract window is first needed for use in the next iteration once we have finished analyzing window $p = q$ using Eq. 3. Notice that in this case by setting $N = 0$ in Eq. 3, it simplifies to W_q with the addition of the initial state constraints. This precisely models the circuit behavior for cycles $q \cdot w$ to $(q+1) \cdot w - 1$ that led to the observed failure. The resulting conflict graph from the SAT-solver will contain clauses that are directly involved in the observed failure at the primary outputs. These clauses, excluding the initial state constraints, form the initial abstraction $AbsW_q$.

In general, the abstraction is formed by maintaining a set of clauses for each time-window that has been previously analyzed. When a time-window has completed analysis, we set $N = 0$ for that instance and clauses within that window are extracted. This is shown in the following equation:

$$AbsW_i = Conflict_i(AbsDebug_i^q(N = 0)) \tag{4}$$

Here, $Conflict_i$ denotes all clauses with variables within time-frames i to $i + w - 1$ involved in the conflict of the input formula. $AbsDebug_i^q(N)$ denotes the abstract debugging instance for the previously analyzed time-window. In the base case for $i = q$, this simplifies to Eq. 1.

Using Eq. 4, the initial abstract debugging problem can be created, shown in the following formula:

$$AbsDebug_p^q(N) = Debug_{p \cdot w}^{(p+1) \cdot w - 1}(N) \wedge \bigwedge_{i=p+1}^{q} AbsW_i \tag{5}$$

This formula mirrors Eq. 3 except it uses the abstract windows for later time-frames instead of explicitly modeling them. It results in a greatly reduced memory footprint because the abstract windows typically are much smaller than the explicit model. In addition, since each abstract window ($AbsW_i$) contains either clauses that were in the explicit model of that window (W_i) or implied by it, Eq. 5 is in fact an over-approximation of Eq. 3 as stated in the next lemma.

Lemma 1 *Let $\mathcal{E}$ be a set of N active suspect variables found in a satisfying assignment to $TimeDebug_p^q(N)$, then there is a satisfying assignment to $AbsDebug_p^q(N)$ that will also contain the active suspect variables from $\mathcal{E}$.*

Lemma 1 implies that the initial abstract instance will return a superset of debugging solutions when compared to explicit modeling. To filter out spurious solutions, each one of them will need to be verified by propagating its satisfying assignment forward to each subsequent concretely modeled time-window. If a solution is indeed spurious, the resulting conflict will act as a refinement step, a process discussed in the following subsection.

3.2 Path Directed Refinement

Due to the use of abstract windows in Eq. 5, a satisfying assignment, $\mathcal{A}$, may not correspond to one in the concrete instance and therefore may be spurious. A straightforward method to verify $\mathcal{A}$ would be to apply it to the entire concrete instance in Eq. 3. However, this would negate the benefit of a reduced memory footprint since the instance would involve explicit modeling of all time-frames. Instead, by utilizing the previous abstract windows, it is possible to create several smaller SAT instances that can equivalently verify $\mathcal{A}$.

In order to propagate $\mathcal{A}$ forward, only a subset of assignments are actually needed. Notice that in Eq. 5, time-frames from $p \cdot w$ to $(p+1) \cdot w - 1$ are modeled explicitly, while the remaining are not. This means that an assignment for the first $p \cdot w$ to $(p+1) \cdot w - 1$ time-frames on the abstract model should also work for the respective frames on the concrete model in Equation 3. However, the only way for these concrete time-frames to affect forward time-frames are through the state variables at time-frame $(p+1) \cdot w$. If these state assignments are used to constrain subsequent concretely modeled time-frames, then we can iteratively propagate the effect of the original assignment forward to each concretely modeled time-window. We denote this subset of assignments of $\mathcal{A}$ on state variables for time-frame i by $cube^i$.

To accomplish this propagation, we create multiple instances each of which models precisely w concrete time-frames and uses the abstract windows for the others. This construction is represented by the following equation:

$$Prop_r^q = cube^{r \cdot w} \wedge W_r \wedge \bigwedge_{i=r+1}^{q} AbsW_i \tag{6}$$

The state assignment $cube^{(p+1) \cdot w}$, extracted from $\mathcal{A}$, is propagated using the above equation to generate $Prop_{p+1}^q$. If this results in SAT, another cube, $cube^{(p+2) \cdot w}$, will be extracted and propagated to the next instance, $Prop_{p+2}^q$, and so on, until all time-frames have been verified. If all subsequent instances result in SAT, then the original abstract satisfying assignment can be extended to one in the the concrete model. This is stated more precisely in the following lemma.

Lemma 2 *Let $\mathcal{E}$ be a set of N active suspect variables found in a satisfying assignment of $AbsDebug_p^q(N)$. If $AbsDebug_p^q(N)$, $Prop_{p+1}^q$, ..., $Prop_q^q$ are SAT, then there is a satisfying assignment to $TimeDebug_p^q(N)$ that will also contain the active suspect variables from $\mathcal{E}$.*

After a set of active suspect variables is found, they are blocked so that the next solution can be found for $AbsDebug_p^q$. Notice that the propagation generates a separate instance for every w time-frames resulting in exactly w concretely modeled time-frames for any given instance. This key point ensures that the memory footprint of the entire process is kept to a minimum.

On the other hand, if $Prop_r^q$ is UNSAT, this means that the original assignment $\mathcal{A}$ cannot be extended to the concrete model. In this case, the abstract window was not sufficiently refined and additional clauses must be added. Similar to the initial abstraction, we extract clauses in the SAT-solver conflict graph and add them to the abstract window. Intuitively, this indicates that additional paths are required. This is shown in the following equation:

$$\hat{AbsW}_r = AbsW_r \cup Conflict_r(Prop_r^q) \quad (7)$$

Notice that the refined abstract window, $\hat{AbsW}_r$, still is an over-approximation of the concrete window because it only contains clauses that are either a subset, or implied by, the concrete window it models.

With the refined abstract window $\hat{AbsW}_r$, we can similarly refine all previous abstract windows. This can be accomplished by re-creating Eq. 6 with $j < r$ and the current refined abstract window ($\hat{AbsW}_r$). Since $\mathcal{A}$ is known to be invalid, each one of these instances will be UNSAT because they are just an extension of the assignment $\mathcal{A}$. After the abstract windows have been updated, the refined formula defined by updating Eq. 5 can be solved for all solutions again. These solutions similarly can be either be confirmed or used to refine the abstract time-windows until the refined instance, $AbsDebug_p^q$, is unsatisfiable.

Once all solutions are found, $\hat{AbsDebug}_p^q$ is UNSAT indicating that debugging has been completed on this time-window. The next theorem confirms the completeness of our approach, that is, the set of solutions found in the abstract model equals to the one found for the concrete model.

Theorem 1 *Let $sols_{abs}$ be the set of confirmed debugging solutions returned by iteratively debugging and refining $\hat{AbsDebug}_p^q$ and let $sols_{time}$ be the set of debugging solutions returned by $TimeDebug_p^q$. If the final refined abstract instance $\hat{AbsDebug}_p^q$ is UNSAT by blocking all solutions in $sols_{abs}$, then $sols_{abs} = sols_{time}$.*

3.3 Overall Algorithm

Algorithm 1 presents pseudo-code for the path directed abstraction and refinement algorithm for time-windows from p to q. The main loop from lines 3-13 iterates through each time-window analyzing them separately. Line 4 constructs the initial abstract instance using Eq. 5 with any previously generated abstract windows ($AbsW_i$). In the first iteration, this equation simplifies to Eq. 1 where the very last time-window is analyzed. The inner WHILE loop (lines 5-11) finds all satisfying assignments and confirms each one by passing the result to the VERIFY procedure. If the assignment is not confirmed, then the procedure refines the relevant abstract windows. Once all the current assignments have been verified, the refined abstract problem is reconstructed (line 10), blocking any solutions that were confirmed from being found again. When the algorithm has finished analyzing the current time-window, it exits the WHILE loop and generates the initial abstraction for the current window on line 12 for use in the next time-window.

The pseudo-code for the VERIFY procedure is also shown in Algorithm 1. The procedure begins by extracting the state cube for window p+1 from the assignment $\mathcal{A}$ on line 17. The outer FOR loop (lines 18-28) propagates the cube forward to subsequent windows using Eq. 6 (line 19). If any of the subsequent time-windows are UNSAT (line 20), then the abstract windows are refined by iterating backwards through the windows (lines 21-24). Each backward iteration reconstructs Eq. 6 with the refined abstract windows on line 22. The refinement step (line 23) extracts clauses from each of these UNSAT instances to update the abstract window. The procedure either returns the confirmed solution (line 29) or will return an empty solution otherwise (line 25).

Algorithm 1 Path Directed Abstraction and Refinement

```
1: procedure PathDebug
2:     sols ← ∅
3:     for p ∈ q, q − 1, ..., 1, 0 do
4:         inst ← AbsDebug_p^q ∧ Block(sols)
5:         while inst is SAT do
6:             Assignments ← SolveAll(inst)
7:             for A ∈ Assignments do
8:                 sols ← sols ∪ Verify(A, p, q)
9:             end for
10:            inst ← AbsDebug_p^q ∧ Block(sols)
11:        end while
12:        AbsW_p ← Conflict_p(inst)
13:    end for
14:    return sols
15: end procedure
16: procedure Verify(A, p, q)
17:    cube^{(p+1)·w} ← ExtractCube(A, p + 1)
18:    for i ∈ p + 1, p + 2, ..., q − 1, q do
19:        inst ← Prop_i^q
20:        if inst is UNSAT then
21:            for j ∈ i, i − 1, ..., p + 2, p + 1 do
22:                inst ← Prop_j^q
23:                AbsW_j ← AbsW_j ∪ Conflict_j(inst)
24:            end for
25:            return ∅
26:        end if
27:        cube^{(i+1)·w} ← ExtractCube(inst, i + 1)
28:    end for
29:    return ExtractSuspect(A)
30: end procedure
```

4. PRACTICAL CONSIDERATIONS

This section presents implementation details that provide significant performance gains to Algorithm 1.

4.1 Leveraging the SAT-solver

There are two main efficiency improvements that leverage the capabilities of modern SAT-solvers. The first improvement involves extensive use of unit assumptions and incremental SAT capabilities of modern solvers [1, 3]. This also requires a minor modification to Algorithm 1. During the VERIFY procedure instead of verifying one assignment at a time, multiple assignments can be verified using one instance of $Prop_i^q$. This is accomplished by setting each cube, $cube^{i \cdot w}$, as unit assumptions and re-solving the instance using incremental SAT. This greatly reduces the overhead of re-solving the same instance (with the exception of the state cube) multiple times. The second improvement makes merging clauses practical by using *clause subsumption* [1]. Since this step is performed each time an abstract window is refined, one can reduce its overhead by using efficient implementations available in many modern solvers.

4.2 Improved Refinement Techniques

Although it will be experimentally shown that the path-based abstraction in Algorithm 1 is memory efficient, in certain cases it may require an excessive number of refinement iterations. We briefly present three additional techniques to manage this issue.

The first technique aims to speed up refinement by attempting to find additional state cubes that may cause a conflict. Multiple additional state cubes can be generated from the original cube ($cube^{i \cdot w}$) by simply inverting a subset of its bits. These additional cubes are applied to Eq. 6 on line 22 of Algorithm 1 using incremental SAT and unit assumptions as described above. If the resulting instance is UNSAT, clauses can be extracted from the conflict-graph

accelerating refinement in an efficient manner.

The next technique re-uses clauses from previous abstract windows. When an initial abstraction for $AbsW_p$ is created on line 12 of Algorithm 1, one can add the equivalent clauses from the previous abstract window, $AbsW_{p+1}$. Note that not all clauses will be valid in the current window due to the differences in the error trace between windows. However, it is efficient to verify if the clause is valid by propagating the negation of the clause using the SAT-solver and the concrete model. The rationale behind this technique is that the same circuit paths may be needed in future abstract windows, so there is no need to duplicate the work by refining them repeatedly.

The last technique handles the worse-case scenario where refinement may require an excessive number of iterations. These cases typically occur when trying to verify certain suspects that require a large number of circuit paths. This exposes itself on line 5 of Algorithm 1 where the WHILE loop takes an excessive number of iterations due to the same suspect appearing in the satisfying assignments. The solution to this problem is to skip these suspects by blocking them if they appear as a solution on line 6 of Algorithm 1 more than *skip* times for a given time-window. *skip* is a user-defined parameter that allows the user to trade-off confirming these suspects for a speed-up in run-time. Practically, the number of suspects skipped is small that it does not have a major impact on quality of results while providing large benefits in run-time.

5. EXPERIMENTS

This section presents the experimental results for the proposed path directed abstraction and refinement algorithm. All experiments are run on a single core of a Intel Core i5 3.1 GHz quad-core workstation with 8 GB of RAM and a timeout of 7200 seconds. Algorithm 1 with the improvements described in Section 4 are implemented and compared against the time-windowing technique in [6] with a time-diagnosis framework. This algorithm named *window expansion* is chosen because it is the only other time-windowing technique that can return exact results through its use of explicit modeling. This is not the case with other existing time-windowing techniques that use approximations with no means of refinement leading to spurious solutions. This time-windowing technique analyzes time-windows starting from the end of the trace. However, it does not approximate suffix time-frames and instead it models them explicitly as in Eq. 3. Note, comparisons to previous abstraction and refinement techniques [7,10] for debugging are omitted because these techniques are orthogonal to the time-windowing techniques presented here. Minisat [3] is used to solve all SAT instances.

The proposed algorithm is exercised on industrial Verilog RTL designs from `OpenCores` [9] and two commercial designs (`fxu`, `comm`) provided by our industrial partners. Each debugging instance is created by randomly selecting a line in the RTL and inserting a typical industrial RTL error (wrong state transition, incorrect operator, erroneous module instantiation, etc.). Such errors typically map to multiple gate-level errors. Next, the buggy design is simulated with its accompanying testbench to expose the failure and record the error trace. After this step, all designs with the inserted error are run through a cone of influence reduction [2] to remove logic in the circuit that is not involved in the debugging problem. This combined with the inserted errors may result in different effective circuit sizes for instances created using the same design. Finally, both algorithms are run for the length of the error trace with error cardinality $N = 1$, time-window size $w = 10$, and $skip = 10$ for the proposed algorithm. The window size w is chosen so that *window expansion* could be run for at least one iteration on the biggest design. The *skip* parameter is chosen to balance the performance and number of skipped solutions over all instances. The solutions returned from both these algorithms correspond to RTL structures such as `if` statements, `assign` statements, instantiations, etc.

The results of the experiments are shown in Table 1. Each row of the table shows results for a separate debugging instance with a different bug. Each instance is labeled by appending the design name with a number to indicate different bugs. Two sets of experiments are run, one for the *window expansion* [6] algorithm, and the other the proposed technique from Algorithm 1.

The first five columns of Table 1 show the instance name, number of gates, number of cycles in the error trace, number of potential suspect locations, and base memory footprint of the design. The base memory footprint is common among both algorithms and includes the memory used to parse the RTL, load the error trace and setup the common circuit data structures. The next four columns show the results for *window expansion* which include the run-time in seconds, peak-memory used for the problem over and above the base memory, the number of solutions returned by the algorithm, and the total number of frames analyzed. The following five columns show results for the proposed method which include run-time, peak-memory for the problem, number of solutions, total number of frames analyzed, and number of skipped solutions. A **TO** (**MO**) entry in the table is used to refer to a time-out (memory-out) condition for that experiment. If a **TO** or **MO** condition occurs, partial results for the corresponding experiments are shown in the table.

The benefit of the proposed technique is apparent by the significant reduction in peak-memory when compared to *window expansion*. The proposed algorithm is able to complete analysis on the entire trace for 78% more instances compared to previous work primarily due to **MO** conditions by the previous technique. This translates to 16 instances for the proposed method versus 9 for *window expansion*. Moreover for instances of Algorithm 1 that ran to completion, the proposed algorithm showed a 55% average reduction in peak problem memory while being able to debug traces that were 31% longer on average. This shows the efficacy of the path-based abstraction to dramatically reduce the memory footprint of large industrial problems that were previously too large to debug.

The run-time results for Algorithm 1 show mixed results compared with *window expansion*. In some cases such as `div64bits_2`, it can perform better. This is primarily because the error propagation path is relatively narrow so very few refinement iterations are necessary. In other cases where both algorithms finished, the run-time can be greater. This is most apparent in `mips_sys_1/2` where the number of paths that needed to be refined was too large causing a time-out. Despite this increase in run-time, it is only comparing instances that finished for both algorithms. Admittedly, the central benefit is that it can solve instances that are too large to fit into memory compared to previous methods.

The number of confirmed solutions returned between both algorithms are approximately the same. In some cases such as `comm_1`, the proposed algorithm was able to find additional suspects that the previous method could not due to a **MO** condition. In other cases, the use of skip suspects reduced the number of confirmed solutions that the path directed abstraction returned. However, the number of unconfirmed suspects is relatively small with 0.36% of total suspects skipped on average. Additionally, the absolute numbers of skip suspects range from 0 for most cases to at most 20, which practically does not decrease the effectiveness of the results for human analysis.

Figure 2 shows a more detailed analysis of the proposed abstraction. This graph shows the relative size of the abstraction in terms of clauses compared with explicit model-

Table 1: Path Directed Abstraction and Refinement Experiments

Instance Info					Window Expansion [6]				Path Directed Abstraction				
instance name	# gates (k)	# total cyc	# total susp	base mem (MB)	time (s)	prob mem (MB)	# sols	# debug cyc	time (s)	prob mem (MB)	# sols	# debug cyc	# skip
ac97_ctrl_1	22.9	1000	1380	278	2641	**MO**	46	610	1886	2478	46	1000	0
ac97_ctrl_2	22.9	1000	1388	278	4077	**MO**	122	550	4023	2571	123	1000	0
comm_1	164.4	150	20155	1453	899	**MO**	72	70	7163	4925	74	150	2
comm_2	164.4	120	20155	1453	1018	**MO**	73	70	4803	3879	75	120	2
div64bits_1	59.7	110	197	347	390	3647	38	110	1929	1122	36	110	8
div64bits_2	57.8	100	198	347	374	3193	21	100	69	873	21	100	0
fdct_1	239.5	80	4662	1302	2450	**MO**	78	50	6673	3611	80	80	3
fdct_2	243.2	70	4661	1302	3304	**MO**	69	50	6414	3154	61	70	8
fpu_1	72.5	240	1982	388	702	**MO**	35	220	312	2024	37	240	0
fpu_2	71.5	40	1981	385	35	1583	5	40	26	674	5	40	0
fxu_1	447.3	100	33087	1872	758	**MO**	174	20	939	**MO**	174	50	0
fxu_2	447.5	100	33087	1872	266	**MO**	37	20	326	**MO**	37	20	0
mips_sys_1	50.3	250	2636	372	2073	7105	123	250	**TO**	2057	104	210	12
mips_sys_2	50.1	200	2636	434	3983	5322	260	200	**TO**	1578	251	140	5
rsdecoder_1	14.5	80	2040	562	5187	1311	81	80	6816	680	78	80	12
rsdecoder_2	14.5	110	2040	489	489	627	384	110	2471	579	384	110	0
usb_funct_1	35.8	410	4061	328	3413	**MO**	105	410	306	1684	105	410	0
usb_funct_2	35.8	690	4061	328	3779	**MO**	106	420	2542	2950	106	690	0
vga_1	48.8	100	1731	624	4955	3545	20	100	3447	1167	20	100	9
vga_2	20.7	150	1729	624	720	1911	46	150	4138	818	46	150	20

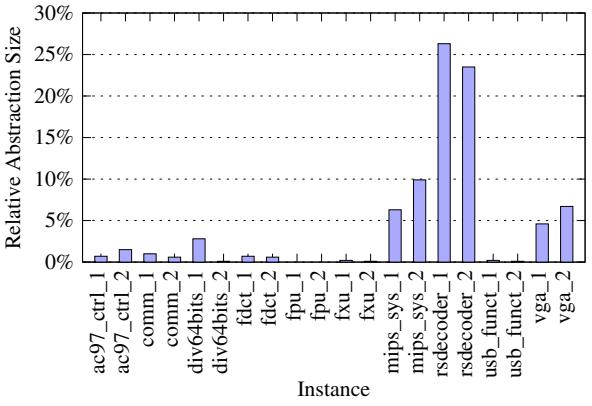


Figure 2: Relative Abstraction Size

ing of those same time-frames. On average, the abstraction is 4.3% the size of the explicit model. Most cases are well below the average indicating that most paths are not needed to solve the debugging instance. However, in some cases such as `rsdecoder_1` the number of clauses is higher at 26%. This translates into less benefit in terms of memory reduction but still a small fraction of the explicit model.

6. CONCLUSION

In this work, a novel path directed abstraction and refinement algorithm is proposed. The algorithm divides the error trace into non-overlapping time-windows where each window is analyzed separately. Subsequent windows are replaced with abstracted over-approximations derived from failing paths in the time domain. Using this abstracted model, each solution found is processed through an additional verification step that removes spurious solutions and simultaneously refines the problem. Experimental results on industrial designs with long error traces show a dramatic decrease in peak memory usage when compared with previous work.

7. REFERENCES

[1] A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[2] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[3] N. Eén and N. Sörensson. An extensible SAT-solver. In *Int'l Conf. on Theory and Applications of Satisfiability Testing*, pages 502–518, 2003.

[4] H. Foster. From Volume to Velocity: The Transforming Landscape in Function Verification. In *Design Verification Conference*, 2011.

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

[6] B. Keng, S. Safarpour, and A. Veneris. Bounded Model Debugging. *IEEE Trans. on CAD*, 29:1790–1803, November 2010.

[7] B. Keng and A. Veneris. Managing complexity in design debugging with sequential abstraction and refinement. In *ASP Design Automation Conf.*, pages 479–484, 2011.

[8] D. McGrath. De Geus touts new products, says ICs will rebound. *EE Times*, March 2009.

[9] OpenCores.org. http://www.opencores.org, 2007.

[10] S. Safarpour and A. Veneris. Automated design debugging with abstraction and refinement. *IEEE Trans. on CAD*, 28(10):1597–1608, 2009.

[11] A. Smith, A. Veneris, M. F. Ali, and A. Viglas. Fault diagnosis and logic debugging using Boolean satisfiability. *IEEE Trans. on CAD*, 24(10):1606–1621, 2005.

[12] Y.-S. Yang, A. Veneris, and N. Nicolici. Automating data analysis and acquisition setup in a silicon debug environment. *IEEE Trans. on VLSI Systems*, PP(99):1–14, 2011.

[13] C. S. Zhu, G. Weissenbacher, and S. Malik. Post-Silicon Fault Localisation Using Maximum Satisfiability and Backbones. In *Formal Methods in CAD*, 2011.

SUPPLEMENTAL MATERIAL

S1. DETAILED EXAMPLES

S1.1 Time Diagnosis Example

This example illustrates time-diagnosis described in Section 2.2. Figure 3(a) visualizes Eq. 3 when $p = q = 1, w = 1$ and $N = 1$. The circuit under debug has three internal gates (g_1, g_2, g_3), one input (x_0), one output (y_0), and three state-variables (s_0, s_1, s_2). In this circuit, g_1 and g_2 form module A and g_3 forms module B. The error models are denoted by $\otimes$. There are two suspect variables, e_a, e_b corresponding to module A and B respectively. The suspect variable e_a, corresponds to the two outputs of g_1 and g_2, similarly for suspect variable e_b and g_3. Error trace values for the initial state, input and expected outputs are shown directly in the figure.

When the instance in Figure 3(a) is passed to a SAT-solver, one solution is returned where $e_b = 1$ corresponding to module B being returned as a suspect. When $\overline{e_b}$ is added as a unit clause to block it from appearing again, the instance is `UNSAT` indicating that no more solutions are found.

Figure 3(b) shows a visualization of Eq. 3 with the next time-window where $p = 0, q = 1, w = 1$ and $N = 1$. In this instance, the suspect variables now only correspond to modules in the earliest time-frame. When sent to the SAT-solver, one solution is returned where $e_a = 1$. After blocking it as a solution, the instance is `UNSAT` indicating that all solutions have been found.

S1.2 Path Directed Abstraction and Refinement Example

This example shows how Algorithm 1 works by using the circuit and error trace from Section S1.1. The first iteration of Algorithm 1 is identical to the instance generated from Figure 3(a). Once this instance returns `UNSAT`, the algorithm sets $N = 0$, sends it to the SAT-solver and extracts clauses involved in the resulting conflict graph to generate $AbsW_1$.

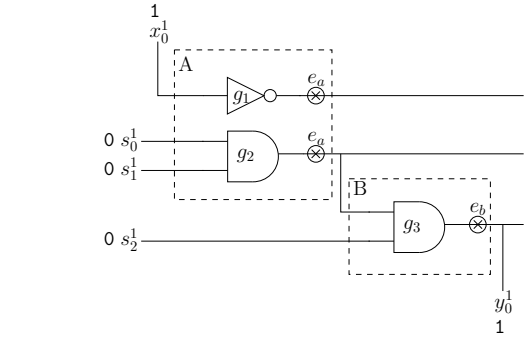
In the next iteration of the main loop, it generates an abstract debugging problem using Eq. 5 with $p = 0, q = 1, w = 1$ and $N = 1$. A visualization of this instance is shown in Figure 4(a). Here, most of circuitry from time-frame 1 has been abstracted leaving only the path directly involved in the conflict from the SAT-solver. Note this path is not unique and another conflict graph could have generated a different path.

When given to the SAT-solver, this instance returns one solution where $e_a = 1$ since e_b was already blocked from the previous time-window. The associated cube with this satisfying assignment is $cube^1 = \overline{s_0^1} \wedge s_1^1 \wedge s_2^1$. This cube is verified by generating a new instance using Eq. 6 with $r = q = 1$, it is shown in Figure 4(b) and one can show that it is `UNSAT`. The clauses from the conflict graph are extracted and the abstract window $AbsW_1$ is refined.

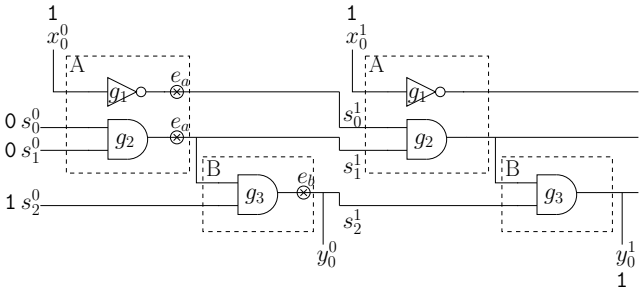
Figure 4(c) shows the refined abstract debugging problem $\hat{AbsDebug}_0^1(N = 1)$. This instance again finds a solution with $e_a = 1$ and a new cube, $\hat{cube}^1 = s_0^1 \wedge s_1^1 \wedge s_2^1$. This cube is verified using Eq. 6 which returns `SAT`. Thus $e_a = 1$ is confirmed as a solution matching the results from Section S1.1. Once this solution is blocked, the refined instance returns `UNSAT` indicating that this time-window has completed analysis.

S2. FORMAL PROOFS

Lemma 1 *Let $\mathcal{E}$ be a set of N active suspect variables found in a satisfying assignment to $TimeDebug_p^q(N)$, then there is a satisfying assignment to $AbsDebug_p^q(N)$ that will also contain the active suspect variables from $\mathcal{E}$.*



(a) $TimeDebug_1^1(N = 1)$



(b) $TimeDebug_0^1(N = 1)$

Figure 3: Time Diagnosis Example

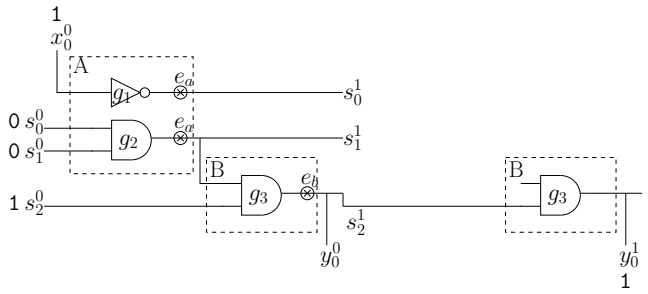
PROOF. Let $\mathcal{A}$ be the satisfying assignment to $TimeDebug_p^q(N)$. We will show that each of the sub-formulas of $AbsDebug_p^q(N)$ is satisfiable under $\mathcal{A}$. $Debug_{p\cdot w}^{(p+1)\cdot w-1}(N)$ is `SAT` under $\mathcal{A}$ because it is a subset of the clauses in $TimeDebug_p^q(N)$. Each $AbsW_i$ is derived from extracting clauses from a SAT-solver conflict graph that are within the same time-frames modeled by W_i. This means that these clauses are either a subset of W_i or implied by the overall problem. Therefore, if all W_i are `SAT` under $\mathcal{A}$, so are all $AbsW_i$. Since each component of is `SAT` under $\mathcal{A}$ so is $AbsDebug_p^q(N)$ with active suspect variables $\mathcal{E}$. □

Lemma 2 *Let $\mathcal{E}$ be a set of N active suspect variables found in a satisfying assignment of $AbsDebug_p^q(N)$. If $AbsDebug_p^q(N)$, $Prop_{p+1}^q$, ..., $Prop_q^q$ are `SAT`, then there is a satisfying assignment to $TimeDebug_p^q(N)$ that will also contain the active suspect variables from $\mathcal{E}$.*

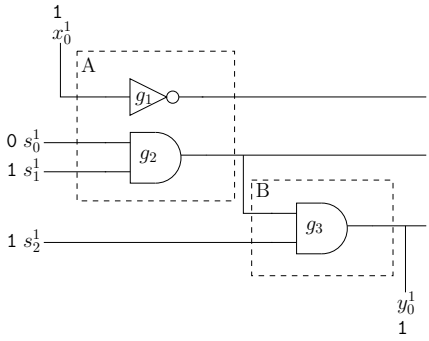
PROOF. Let $\mathcal{A}_p, \mathcal{A}_{p+1}, \ldots, \mathcal{A}_q$ be the satisfying assignments to $AbsDebug_p^q(N)$, $Prop_{p+1}^q, \ldots, Prop_q^q$, respectively. We show how to construct an assignment $\mathcal{A}$ to $TimeDebug_p^q(N)$ from $\mathcal{A}_p, \mathcal{A}_{p+1}, \ldots, \mathcal{A}_q$.

From $\mathcal{A}_p$, we add all assignments to variables involved in the subformula $Debug_{p\cdot w}^{(p+1)\cdot w-1}(N)$ to $\mathcal{A}$. From each of the subsequent $\mathcal{A}_r$, we add assignments to all variables of W_r from the respective instance $Prop_r^q$. Notice the overlapping variables that were added to $\mathcal{A}$ are precisely the state variable from the cubes $cube^{p\cdot w}$, $cube^{(p+1)\cdot w}$, ..., $cube^{q\cdot w}$. But from the formulation of $Prop_{r-1}^q$, each cube, $cube^{r\cdot w}$, is generated from an instance that is constrained by the previous cube, $cube^{(r-1)\cdot w}$, except for the first one derived from the assignment $\mathcal{A}_p$. This means that there is no conflict between these overlapping state variables because each one is implied by the previous one in the sequence which originates from $\mathcal{A}_p$.

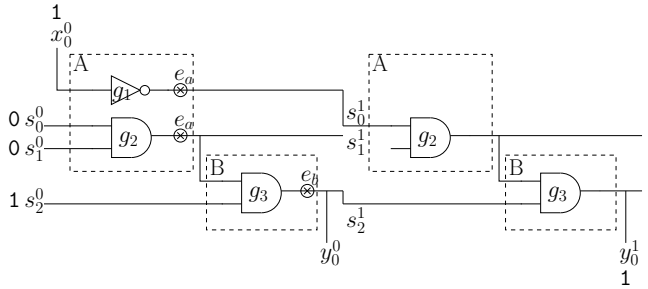
The final constructed assignment $\mathcal{A}$ composes a full assignment to $TimeDebug_p^q(N)$ since it contains assignments to all variables in each component of the formula. Moreover,



(a) $AbsDebug_0^1(N=1)$



(b) $Prop_1^1$



(c) $Abs\hat{D}ebug_0^1(N=1)$

Figure 4: Path Directed Abstraction and Refinement Example

$\mathcal{A}$ is a valid satisfying assignment to each of the components of $TimeDebug_p^q(N)$. We can see this because we constructed $\mathcal{A}$ by extracting exactly the assignments involved in the concrete parts of the abstract formulas which are identical to the concrete model. These concrete parts compose exactly the clauses of $TimeDebug_p^q(N)$, so it too is `SAT` under $\mathcal{A}$ with active suspect variables $\mathcal{E}$. □

Theorem 1 *Let $sols_{abs}$ be the set of confirmed debugging solutions returned by iteratively debugging and refining $Abs\hat{D}ebug_p^q$ and let $sols_{time}$ be the set of debugging solutions returned by $TimeDebug_p^q$. If the final refined abstract instance $Abs\hat{D}ebug_p^q$ is* `UNSAT` *by blocking all solutions in $sols_{abs}$, then $sols_{abs} = sols_{time}$.*

Proof. From Lemma 1, we know that $sols_{abs} \supseteq sols_{time}$. From Lemma 2, we also know that any set of debugging solutions $\mathcal{E}$ found and verified by $Abs\hat{D}ebug_p^q$ is also valid for $TimeDebug_p^q$ *i.e.,* $sols_{abs} \subseteq sols_{time}$. Therefore, $sols_{abs}$ is both a superset and subset of $sols_{time}$, so $sols_{abs} = sols_{time}$. □