Abstraction and Refinement Techniques in Automated Design Debugging

Sean Safarpour, Andreas Veneris
Department of Electrical and Computer Engineering
University of Toronto, Toronto, Canada
{sean, veneris}@eecg.toronto.edu

***Abstract*— Verification is a major bottleneck in the VLSI design flow with the tasks of error detection, error localization, and error correction consuming up to 70% of the overall design effort. This work proposes a departure from conventional debugging techniques by introducing abstraction and refinement during error localization. Under this new framework, existing debugging techniques can handle large designs with long counter-examples yet remain run time and memory efficient. Experiments on benchmark and industrial designs confirm the effectiveness of the proposed framework and encourage further development of abstraction and refinement methodologies for existing debugging techniques.**

I. Introduction

Functional verification of today's VLSI designs is a critical and time consuming task. The processes of verifying the functional correctness of a design, determining the source of the potential error(s) and correcting those errors, can take up 70% of the overall design time [1], [2]. While there exists a plethora of methodologies for verification (i.e. error detection), there is fewer work dedicated towards debugging, that is error localization and correction [3].

Today, the verification and design engineers have the daunting and tedious task of analyzing the design, its specifications and the incorrect response from the simulation traces (counter-examples) to determine the source of errors. For real-life industrial designs such as microprocessors and DSP components, experience shows that traces can often be over tens of thousands of clock cycles long, a fact that makes debugging even more challenging [1], [4]. It is reported that finding the source of error(s) can take up to 50% of the overall verification task, a considerable contributor to the verification bottleneck [1]. Therefore, cost-effective automated debugging methodologies are of great importance to the academic and industrial communities.

Currently, automated design debugging approaches are based on simulation, symbolic, or constraint satisfaction techniques [5], [6], [7]. Most of these approaches use information from the erroneous design, the input logic values, and the expected output logic values to return a set of suspect gates [7]. In sequential design debugging, the circuit representation is often replicated or unrolled for all clock cycles to model the error through time [6]. Clearly, this debugging practice can result in excessive memory and run-time requirements even for modest size designs with hundreds of clock cycle traces.

Reducing the memory demands for sequential debuggers is critical in making existing debugging methodologies practically viable. Current memory reduction techniques partition the problem into subproblems that are solved sequentially [7], they trade time for space by formulating the problem as a Quantified Boolean Formula satisfiability instance [8], and they reduce the length of the traces using formal techniques [9], [10]. Although these methods can be effective in decreasing memory requirements, the problem of debugging large industrial designs remains intractable.

This work introduces a new methodology for design debugging based on the formal concepts of design abstraction and refinement [11], [12]. It does not propose a new debugging method but it presents a novel framework that existing debugging techniques can utilize. Under this new framework, an abstract model of the design is first created to undergo debugging. Since this representation contains less logic than the original one, the size of the problem may be reduced considerably in favor of debugging. The benefits are reduced memory requirements and potentially shorter run times when compared to debugging the original design. Since the abstract model contains fewer state elements than the original one, it may lead to shorter traces when state matching trace reduction techniques are used [9]. Debugging an abstract model can sometimes return abstracted state elements as error sources. In these cases, a refinement procedure is proposed that replaces some of the abstracted variables with the original state elements. Furthermore, the proposed debugging methodology guarantees correctness, where the solutions found under the framework are also solutions in the concrete design. Finally, the completeness of the framework is ensured as no error sources remain undetected for a given set of test vectors.

Given the impact of abstraction and refinement techniques in model checking, it is natural to expect similar results in design debugging. Indeed, experiments confirm the practical benefits of the proposed framework as considerable memory reductions of over 60% and speed-ups of over 4.5X are observed on a set of benchmark and industrial designs while preserving the resolution. These results encourage further research in abstraction and refinement methodologies as an aid to existing debugging techniques.

This paper is organized as follows. Section II provides background information relating to debugging as well as abstraction and refinement. The proposed abstraction and refinement debugging framework is presented in the Section III. The empirical results are provided in Section IV while Section V concludes this work.

II. Preliminaries

This section presents terminology used in the paper. It provides a brief background on design debugging using Boolean satisfiability (SAT) as well as an introduction to abstraction and refinement techniques in model checking. Although SAT-based debugging is used to explain various theoretical concepts in this paper and is used as the debugging engine in the experiments, the proposed framework is not confined to any unique aspects of this debugging technique.

A. Background

Given a set of vectors V for which a circuit (or netlist) C demonstrates an incorrect behavior, the objective of design debugging is to find the gates that may be responsible for this incorrect behavior [5]. In the context of this paper, the set of vectors V include the initial state value, the sequence of primary input values and the *correct* or *expected* primary output values for every clock cycle or time frame. In other words, the specifications for the erroneous circuit act as a "black box" without knowledge about its internal structure. The set V can be derived from a simulation trace or from a formal

verification engine. The terms vector, trace, and counter-example are used interchangeably in this paper.

A circuit C is composed of a set of primary inputs $x_1, x_2, ...$, primary outputs $y_1, y_2, ...$, primitive gates $l_1, l_2, ...$, and state elements $q_1, q_2, ...$ such as flip-flops or latches. An interconnect is referred to by the name of its driving gate. For example, the wire connecting the output of gate l_i to an input of gate l_j is simply referred to as l_i.

B. Debugging with Constraint Satisfaction

The constraint satisfaction problem in SAT-based debugging is generated by adding extra logic to the erroneous circuit C, converting the new circuit into Conjunctive Normal Form (CNF), replicating and constraining the CNF for every vector and time frame in V [7]. The constraint or CNF problem is solved by a SAT solver which returns a set of gate locations where a *correction* (function change) can produce the expected outcome captured in V.

To model the corrections in C, a multiplexer m_i is added for every gate (and primary input) l_i. The output of this multiplexer, m_i, is connected to the fanouts of l_i while l_i is disconnected from its fanouts. This construction has the following effect: when the select line s_i of a multiplexer is inactive ($s_i = 0$), the original gate l_i is connected to m_i, otherwise, when $s_i = 1$ a new unconstrained primary input w_i is connected. Figure 1 illustrates the above transformation for a combinational circuit.

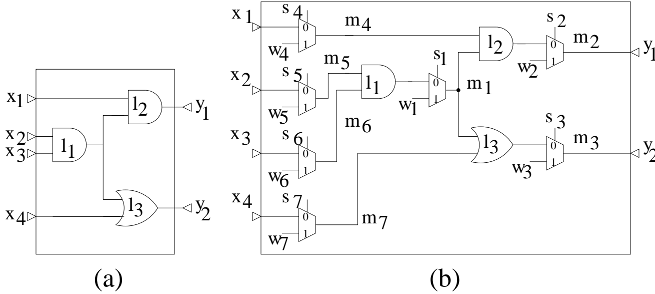

Fig. 1. Extra logic for SAT-based debugging

A potential correction on line l_i is indicated when the select line s_i is assigned to 1 under which condition the correction value is stored in w_i. The SAT solver can assign any value {0,1} to the s_i and w_i variables such that the CNF satisfies the constraints applied by the vectors V. To force the SAT solver to find a specific number N of error locations, further logic is added to activate at most N select lines. Thus for $N = 1$, a single s_i is set to 1 which *corresponds* to candidate error location l_i. In the following, an erroneous gate location is referred to as an error source and an error stemming from N distinct locations is called an N-tuple error.

It is known that many *equivalent* errors may exist for a set of vectors and for a fixed design error model [6]. Intuitively, this is true because there may be more than one way to synthesize and correct a design. In this paper, we say that the debugging procedure is not *complete* unless all equivalent error sources are found. This is performed iteratively by finding a solution to the CNF problem, adding it subsequently to the CNF as a blocking clause and solving the problem again until no more solutions are found.

C. Abstraction and Refinement

Abstraction and refinement techniques are used readily in model checking to mitigate the exponential nature of the underlying state space [11], [12], [13]. Roughly speaking, an *abstract model* is derived by removing some state elements from the original or *concrete* design using some abstraction function $\hbar$. The reduced number of state elements result in fewer states to consider when verifying properties.

An abstract model can be derived by the following simple steps:

1) Use the abstraction function $\hbar$ to remove some state elements from the concrete design.

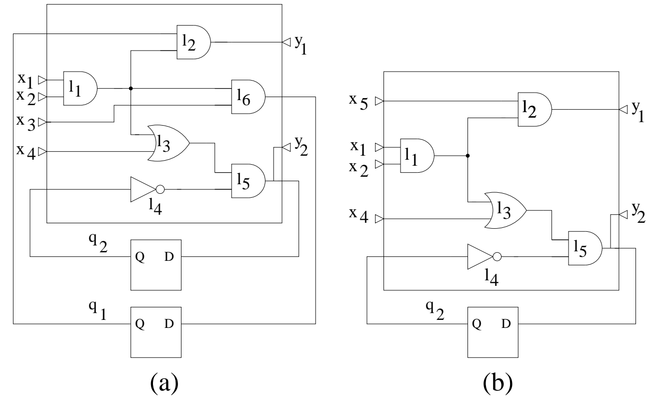

Fig. 2. Abstracting q_1 from a circuit

2) Remove all the combinational logic that are only in the transitive fanin of the abstracted state elements.
3) For each removed state element, introduce a primary input and connect it to the fanout of a removed state element.

For example, Figure 2 (a) and (b) illustrate a circuit before and after abstracting the state element q_1, respectively.

For safety properties, if model checking determines that a property holds in the abstract model, then it must also hold in the concrete design [11]. However, if a property does not hold in the abstract model, then the corresponding counter-example may or may not hold in the concrete design. If the counter-example is not valid on the concrete design it is said to be *spurious* [11]. In this case, the abstract model is *refined* by reverting some of the abstracted state elements and continuing the model checking process.

III. Debugging with Abstraction and Refinement

This section proposes a new framework for debugging using abstraction and refinement. The initial formulation is presented in Section III-A. In Section III-B, the occurrence of unjustifiable solutions is elaborated and the methodology is re-formulated to prevent them. In Section III-C, this framework is extended for completeness.

A. Basic Construction

The abstraction and refinement methodology first creates an abstract model. This model and its corresponding set of vectors V are used to generate a new debugging problem instance. If debugging is formulated as a constraint problem described in Section II-B, the problem is solved by a SAT solver. Next, the solutions returned by the SAT solver must be verified on the concrete design to determine whether they are unjustifiable or spurious. Spurious solutions are used to refine the abstract model and the process is repeated with the new model. In the following, we describe this process in detail.

An abstract model C' is constructed by removing (i.e. abstracting) the state elements and replacing them with primary inputs. Which and how many state elements are selected for removal is determined by the abstraction function $\hbar$. Once the state elements are abstracted, all logic in their transitive fanin is also removed.

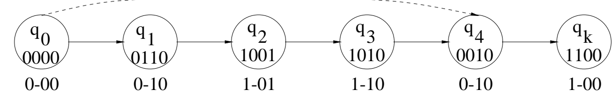

Fig. 3. Reduced trace V' due to abstraction

Since the abstract model may have fewer state elements than the concrete model, a set of more compact traces V' can be obtained through state matching [9], [10]. State matching procedures can remove redundant state transitions between repeated states or they can employ formal techniques to determine the minimum trace length achievable.

Algorithm 1 Basic Abstraction and Refinement Debugging

```
1:  Solutions = ∅
2:  C' = abstract(C)
3:  while (1) do
4:     V' = trace_reduction(C', V)
5:     New_sols = debug(C',V')
6:     if ( New_sols = ∅) then
7:        return Solutions
8:     end if
9:     Solutions = Solutions ∪ New_sols
10:    for all Sol ∈ New_sols do
11:       if ( unjustified_solutions(Sol, C, V')) then
12:          Solutions = Solutions \ Sol
13:       else if ( spurious_solutions(Sol, C')) then
14:          C' = refine(Sol,C')
15:          Solutions = Solutions \ Sol
16:       end if
17:    end for
18: end while
```

As an example consider Figure 3 where a state transition diagram is used to illustrate an error trace from state q_0 to q_k. In the original trace, no trace reductions are possible through state matching. However, after the second state element is removed (through abstraction) the states q_1 and q_4 can no longer be differentiated. The state values after abstraction are shown under each node in Figure 3. As a result, a "short-cut" can be taken in the trace from state q_0 to state q_4 as illustrated by the dashed line. Similar to many trace reduction techniques, the compacted traces V' must be tested to determine whether the error(s) are still observable [10].

Next, the abstract model and the compacted traces V' are used to formulate the debugging problem. Since the abstracted model contains less logic and has potentially shorter traces than the concrete design, the size of the CNF is expected to be smaller in terms of clauses and variables and thus require less memory. As described in Section II-B, the SAT solver returns a set of location tuples which are potential error sources. These potential error sources represent locations where a correction (*i.e.,* some function change) can be applied which results in a correct behavior of the abstracted model as dictated by the reduced set of vectors V'. The SAT solver also assigns logic values to all abstracted variables which are now unconstrained primary inputs in the abstract model.

Although viable for the abstract model, the solutions returned by the SAT solver may not be viable for the concrete design. More specifically, the logic value assignments made to the abstract variables by the solver may not be justifiable in the concrete design where these variables are constrained by their original fanin logic.

Definition 1 *Debugging an abstract model results in an* unjustifiable *solution if the logic value assignments cannot be justified for the corresponding variables in the concrete design.*

To verify whether the solutions are justifiable, a constraint problem is formulated using the concrete design, C, the solutions returned by the debugger, and the reduced traces V'. This problem is provided to a SAT solver where an unsatisfiable result determines that the abstracted variable logic assignments are unjustifiable according to Definition 1.

Definition 2 *Debugging an abstract model results in a* spurious *solution if any of the error tuples returned correspond to an abstracted variable.*

According to Definition 2, spurious solutions do not provide enough information about the error sources since the abstracted variables have their fanin logic removed from the original design. In this case, the model is *refined* using these abstracted variables so that the non-abstracted error sources can be found. The debugging continues with this newly refined model until all equivalent error sources are found.

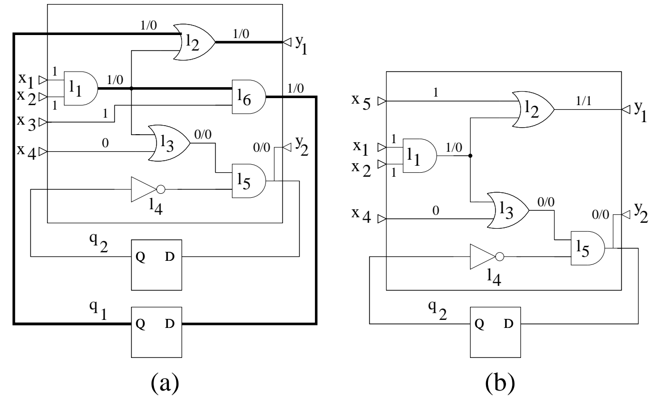

Fig. 4. Effect of unconstrained variables

Algorithm 1 illustrates the basic abstraction and refinement procedures described thus far. On lines 2 and 4, the initial abstract model C' and the reduced set of vectors V' are generated. Line 5 calls the debugger to search for solution tuples while the unjustifiable solutions are removed on line 12. For the remaining solutions, spurious ones are filtered out and the abstract model is refined as shown on lines 14-15. This process is repeated until no new solutions are found.

As mentioned earlier, algorithm 1 is not restricted only to SAT-based debugging. For instance, Binary Decision Diagram (BDD) based debugging methods [6] can be employed by building a BDD representation of the abstract circuit. Similarly, simulation-based techniques [6], can perform simulation and path-tracing on the abstract model as they search for solutions.

B. Guaranteeing Correctness

In the formulation of Section III-A, by leaving the abstract variables unconstrained in the problem CNF, the SAT solver may trivially assign them logic values so that the erroneous abstract model produces the correct response. In fact, the SAT-based debugging formulation of Section II-B may be satisfied without activating any of the multiplexer select lines (i.e. $\forall i, s_i = 0$) or when $N = 0$. The following example illustrates this situation.

Example 1 *Figure 4 (a) shows a concrete design with an error on gate l_1 which forces the output to 0. The correct/erroneous value of 1/0, shown in bold, propagates from gate l_1 through the flip-flop q_1 and to the primary output y_1. Notice that the primary input values remain constant for both time frames. When the state element q_1 is abstracted and left unconstrained, the SAT solver can assign this new input x_5 to 1 which will produce the correct/erroneous outcome 1/1 as shown in Figure 4 (b).*

The above example shows that the SAT solver can satisfy the problem without activating any multiplexer select lines. As a result, when $N \geq 1$, some solutions returned by the SAT solver are *unjustifiable* as stated by Theorem 1 below.

Theorem 1 *There exist automated debugging problem instances with unconstrained abstracted variables such that the solutions to the problems are unjustifiable.*

Proof: As shown by Example 1, leaving abstracted values unconstrained can result in satisfying the constraint problem with $N = 0$. Since the debugger finds locations that can correct the abstract design with $N \geq 1$, all N-tuple locations in the design qualify by setting any s_i variable to 1 and assigning the value of the l_i variables to the w_i variables. In effect the solutions simulate the behavior of the abstract circuit which is error-free under the current variable assignment. Since there is an error in the erroneous circuit by definition, the solutions for the abstract model must be unjustifiable in the concrete design. ■

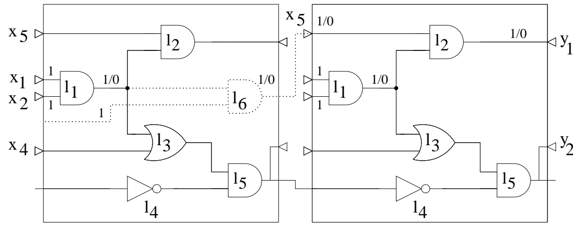

Fig. 5. Abstract model for two time frames

The consequence of Theorem 1 is that the process of determining whether solutions are unjustifiable, presented in Algorithm 1, may not be practical since all N-tuple solutions must be verified to work on the concrete design. Theorem 2 states that unjustifiable solutions can be prevented altogether.

Theorem 2 *Constraining abstracted variables to the values of the corresponding concrete state elements from sequential simulation using the initial state and input values in V' prevents unjustifiable solutions from occurring.*

Proof: The objective of this proof is to show that the abstract model can be restricted sequentially to behave like the concrete model and thus prevent unjustifiable solutions from occurring. There exist a well defined sequence of state transitions from the initial state to the state where the erroneous behavior is witnessed that can be observed through the simulation of C with the input and initial state vectors V'. By applying the logic values of the concrete state elements onto the corresponding abstracted variables, the latter will be constrained to the values in V'. As a result, both the concrete design and the abstract model will contain the same constraints except that one is enforced by logic circuitry while the other is enforced by logic values. Since unjustifiable solutions do not occur in the concrete design by definition, they will not occur in the abstract model either. ■

By Theorem 2, unjustifiable solutions are prevented by constraining the abstracted variables to the values of their concrete counterparts. Thus *correctness* is guaranteed since all solutions for the abstract model are also solutions for the concrete design. Next, the proposed methodology is extended to guarantee that all equivalent error sources are found.

C. Guaranteeing Completeness

SAT-based debuggers such as those described in Section II-B can find all actual and equivalent errors for a given value of N. In the methodology described in Section III-B, it is not the case that a single gate-level error is found at $N = 1$. In other words, a set of m errors in the concrete design may be mapped onto a set of n errors in the abstract model, where $n > m$, as the following example illustrates.

Example 2 *Consider the abstract circuit in Figure 2 (b) unfolded over two time frames as illustrated in Figure 5. For clarity, the abstracted logic l_6 is shown in dashed lines. Notice that the error from gate l_1 does not directly propagate to output y_1 but its effect is captured in the abstract variable x_5. For $N = 1$ the SAT solver returns the single equivalent error location l_2. Assuming that the design is analyzed and it is concluded that l_2 is not the error source, the real source of error goes undetected. However, if N is incremented to 2, then the pair l_1 and x_5 is found as a solution. By refining the abstract variable x_5 to q_1 and solving the debugging problem again with $N = 1$, the single error location l_1 is found.*

Theorem 3 states that the process outlined in Example 2 finds all equivalent error locations and is thus complete.

Theorem 3 *The debugging procedure that performs the following steps is complete for some value of $maxN$.*

1) *Perform debugging for N-tuple errors using the abstract model*
2) *If an abstracted variable is returned as an error location, refine the model and set $N = 0$*
3) *Increment N by 1*
4) *Go to (1) unless $N > maxN$*

Algorithm 2 Complete Abstraction and Refinement Debugging

1: *Solutions* $= \emptyset$, $N = 1$
2: C' =abstract(C)
3: **while** (1) **do**
4: V' =trace_reduction(C', V)
5: *Const* = extract_constraint(C', V')
6: *New_sols* = constrain_and_debug(C',V',N,*Const*)
7: *Solutions* = *Solutions* $\cup$ *New_sol*
8: **for all** *Sol* $\in$ *New_sols* **do**
9: **if** (spurious_solutions(*Sol*, C')) **then**
10: C' =refine(*Sol*, C')
11: *Solutions* = *Solutions* $\setminus$ *Sol*
12: $N = 0$
13: **end if**
14: **end for**
15: $N = N + 1$;
16: **if** ($N > maxN$) **then**
17: return *Solutions*
18: **end if**
19: **end while**

Proof: Since at some point N will equal the number of errors mapped in the abstract design n, all the equivalent errors that map into n-tuples or fewer error sources will be found. If any of these locations correspond to abstract variables, then the abstract model is refined and those variables are replaced with their corresponding concrete state elements. The new abstract model is then provided to the debugger which starts the search with $N = 1$. Since some previously abstracted variables no longer exist in the new abstract model, previous solutions at $N = n$ will be found at $N \leq n$. This process continues until no new solutions are found which guarantees that all error sources are found for $maxN = n$, an event that guarantees completeness. ■

Empirical results from Section IV show that for single errors, $maxN = 3$ is large enough to find all equivalent error locations. Furthermore, not much time is spent on debugging problems for which there exists no solutions for a particular N. Algorithm 2 illustrates the overall abstraction and refinement based debugging methodology that guarantees correctness and completeness. It is similar to Algorithm 1 with the following differences. On lines 5 and 6 simulation values are extracted and used to constrain the abstract variables. Spurious solutions are refined and N is reset on lines 10-12 while N is incremented and checked to be less than $maxN$ on lines 15-17.

IV. Experiments

This section presents the experiments conducted to evaluate the effectiveness of the proposed framework. The debugging problems are generated using a sample of four ISCAS'89 circuits, four ITC'99 circuits and four industrial circuits from OpenCores.org [14]. The erroneous circuits are created by changing the type of a single gate at random. An average of 10 traces are obtained per problem through pseudo-random simulation of the correct and buggy circuits until a different outcome is observed. The automated debugger used is a sequential SAT-based debugger similar to [7]. The experiments are conducted on a 2.66GHz Intel Xeon processor with 2 GB of memory with a timeout of 7200 seconds for each SAT problem.

Before starting the debugging process, a simple trace compaction procedure is always performed. This procedure first builds a graph of the visited states, it then connects edges between repeated states and applies Dijkstra's shortest path algorithm from the initial state to the final state [15]. More effective trace compaction algorithms can be applied for better results [9], [10]. After the trace compaction, all traces are used as simulation stimulus to ensure that they still exhibit a different result for the correct and buggy designs.

In summary, Figure 6 shows the effects of abstraction on the logic size and trace lengths. Part (a) demonstrates that the logic size reductions appear to be linear with respect to the number of abstracted state elements. However, in part (b), experiments show that significant trace length reductions are not observed until a certain threshold is

TABLE I

PROBLEM INFORMATION AND STATISTICS FOR STAND-ALONE SAT-BASED DEBUGGING APPROACH

circuits	# gates	# FF	# clk	# red. clk	# cls (K)	mem (MB)	time/err (s)	# err	total (s)
b04	711	66	516	335	2422	1132	740.0	9	6660.0
b08	200	21	21	20	274	82	3.8	4	15.2
b12	1140	121	40	19	1492	449	165.9	5	829.5
b14	6028	245	54	54	memout	> 2000	-	-	-
s1488	693	6	104	5	214	42	1.6	9	14.4
s5378	3222	179	3	3	554	105	13.1	3	39.3
s13207	9442	669	2	2	1415	227	70.1	9	630.9
s35932	21147	1728	75	8	3563	696	431.1	16	6897.6
div_su	1528	126	9	6	607	109	12.4	64	793.6
rsdecoder	10629	521	2	2	2043	301	120.1	9	1080.9
spi	2027	90	20	18	2763	582	391.3	3	1173.9
ac97	15166	1452	30	30	memout	> 2000	-	-	-

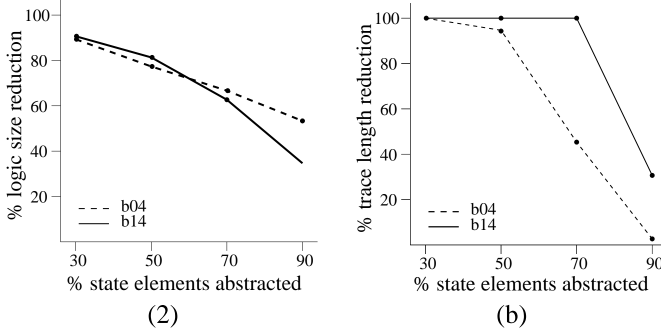

Fig. 6. Logic and trace reduction vs. flip-flops abstracted

reached. This threshold appears to be over 50% for b04 and over 70% for b14. Thus for large problems where memory is a major concern, a more aggressive approach, where over 70% of state elements are abstracted, may be desirable.

Table I presents a summary of the debugging problems as well as some insightful performance statistics for debugging the concrete circuits using the conventional SAT-based debugger. Later, these results are contrasted with those of the proposed debugging framework. Columns 1, 2 and 3 present the circuit name, number of gates, and number of flip-flops (state elements) in each circuit. Columns *clk* and *red. clk* show the average length of the traces before and after the trace compaction, respectively.

The next five columns summarize the results of the debugger for each problem. In Columns *# cls* and *mem*, the number of clauses in thousands generated for each problem and the debugger's memory usage is presented. The number of equivalent errors found by the debugger for the given vectors as well as the average time required to find them are presented in columns *# err* and *time/err*, respectively. Finally, the total time required to find all the errors is presented in column *total*.

To cope with the size of the larger problems the CNFs are partitioned into bands and solved sequentially as described in [7]. For b14 and ac97 where the average reduced traces are 54 time frames and 30 time frames long, the problems still run out of memory. The proposed framework that uses abstraction is most beneficial for such memory intensive problems.

Table II presents the results of the proposed abstraction and refinement SAT-based debugging framework. For each problem, a random abstraction function is used such that between 40-50% of the state elements are abstracted, a conservative amount according to Figure 6. To allow easy comparison with Table I, the percentage of reduced logic, reduced of flip-flops, additional compacted traces, and overall reduced memory requirements are presented in columns 2-5, respectively. Looking across one row for problem b08, by abstracting 47.6% of the flip-flops, the logic is reduced by 26% and the trace length is reduced by an additional 65% which leads to an overall memory reduction of 60% versus the stand-alone debugger.

The largest problems in Table I, b14 and ac97, which ran out of memory previously are both solved under the proposed framework. On the average, the proposed methodology results in up to 60% memory reduction with average savings of 30% under a conservative abstraction approach.

The majority of problems in Table II do not benefit from additional trace compaction. This can be attributed to the fact that trace reduction is most effective for long traces since the probability of matching states is higher. In the experiments, the initial trace compaction process is able to reduce the traces considerably. For instance, the initial trace of circuit s1488 which is 104 clock cycles is reduced to only 5 clock cycles after compaction, thus further reductions are highly unlikely. For industrial traces of thousands of clock cycles derived from functional testbenches and not randomly, it is highly unlikely to reduce traces drastically by simple state matching techniques [9]. Therefore, trace reduction via abstraction may be more effective.

A summary of the run time results of the proposed framework is presented in columns 6-12 of Table II. In columns *time/err* and *# err* the average time required to find an error and the number of errors found are presented, respectively. It should be noted that when the number of errors are greater than those in Table I, it means that abstracted state variables are found as errors. In these experiments, if all equivalent error tuples are found (including the actual inserted error), then no refinement is performed. Note that in practice, not all equivalence errors are necessary since only the actual error is of interest to the designer. If the errors found by the proposed framework do not include all equivalent error locations (i.e. *# err* is smaller in Table II than Table I), then all spurious solutions must be refined.

In Table II, column *maxN* shows the maximum number of tuples searched until all equivalent errors are found. The debugging time for all searches prior to *maxN* is shown in the column *prev*. When refinements are necessary, the column *refine* presents the solve time for all subsequent refinement searches.

For many problems in Table II, the maximum error tuple found (*maxN*) is often greater than 1 but always less than or equal to 3. This signifies that a single error in the concrete design maps to 3 or fewer locations in the abstract model. The time required to determine that no solutions exist prior to *maxN* (*prev*) is always quite smaller than the average time required to find an error (*time/err*). Take b12 for instance, it takes on average 4.2 seconds to determine that no errors occur when $N < 3$ and 85 seconds to find each solution at $N = 3$. Relating these times to Algorithm 2, it means that the approach is quite effective since the majority of the time is spent in `constrain_and_debug` occurs when N=*maxN* and not when $N <$ *maxN*.

The total debugging time for the proposed approach is found by summing the product of *time/err* and *# error* with *prev* and *refine*. The resulting total run time is shown in column *total* and its improvement over Table I is shown in column *X impr.* When abstracting 40-50% of the state elements, not many refinement steps are necessary as most

TABLE II

PERFORMANCE STATISTICS FOR ABSTRACTION AND REFINEMENT DEBUGGING FRAMEWORK

circuits	red. logic(%)	red. FF(%)	red. trace(%)	red. mem(%)	time/err (s)	# err	maxN	prev (s)	refine (s)	total (s)	X impr.
b04	20.5	45.4	0	**9.8**	530.0	12	3	11.0	0	6371	**1.04**
b08	26.0	47.6	65.0	**60.0**	0.2	12	3	0.1	0	3.35	**4.53**
b12	26.4	41.3	15.7	**24.9**	85.0	20	3	4.2	0	1704.2	**0.48**
b14	15.3	40.8	0	$>$ **46.0**	3740.2	2	2	42.0	0	7522.4	**-**
s1488	20.4	50.0	0	**11.9**	1.1	9	1	0	0	9.9	**1.45**
s5378	9.7	44.6	0	**37.1**	11.8	1	1	0	3.4	15.2	**2.58**
s13207	29.6	44.8	0	**31.7**	40.3	9	1	0	0	362.7	**1.73**
s35932	31.9	46.2	0	**34.9**	251.3	16	2	7.3	0	4028.1	**1.71**
div_su	34.0	39.6	0	**9.5**	5.9	32	3	2.2	396.8	587.8	**1.35**
rsdecoder	34.7	43.1	0	**22.9**	54.8	7	1	0	0	383.6	**2.81**
spi	37.6	44.4	22.2	**46.0**	101.2	1	1	0	303.6	404.9	**2.89**
ac97	41.2	48.2	0	$>$ **37.0**	365.6	2	1	0	0	731.2	**-**

TABLE III

SUMMARY OF B14 WHEN ABSTRACTING OVER 80% OF FLIP-FLOPS

step	red. logic(%)	red. FF(%)	red. trace(%)	mem(MB)	time/err(s)	err
Tbl II	15.4	40.8	0	1080	3740.0	2
abs	52.7	81.6	20.3	344	172.0	4
ref 1	50.2	80.8	20.3	378	225.1	3
ref 2	50.1	80.4	20.3	404	242.3	10

TABLE IV

SUMMARY OF AC97 WHEN ABSTRACTING OVER 96% OF FLIP-FLOPS

step	red. logic(%)	red. FF(%)	red. trace(%)	mem(MB)	time/err(s)	err
Tbl II	41.2	48.2	0	1260	1567.8	2
abs	89.7	96.4	33.3	555	365.6	2
ref 1	89.5	96.3	33.3	765	665.8	10
ref 2	89.4	96.2	33.3	773	664.0	6
ref 3	89.1	96.1	33.3	776	721.8	9

equivalent error locations are found in the abstract model. However, even for the cases where refinement is necessary, substantial run time improvement is observed. The only problem that demonstrates a performance decrease is b12 where four times more solutions are found in the abstract model versus the concrete design. Overall, performance improvements of up to 4.5X are observed with an average value of 2X across all problems. This increased efficiency can be attributed to the smaller size of the constraint problems which lead to easier CNFs for the SAT solver.

As observed in Figure 6 smaller problem sizes and shorter traces can be achieved with more aggressive abstraction than those of Table II. To demonstrate the effectiveness of the framework under a more aggressive abstraction strategy, the two largest problems b14 and ac97 are shown in Tables III and Tables IV with 80% and 96% of the state elements abstracted. For easy comparison, the first row of each table re-presents the problem properties of Table II. The following rows show the results after each abstraction and refinement steps until the specific injected error is found (not all equivalent errors as in Table I). For each table, column 1 describes whether the data is derived from Table II (*Tbl II*), from the initial abstraction (*abs*), or from a refinement step (*ref*). The remaining columns are labeled similarly to Table II.

As expected, when more state variables are abstracted, greater memory saving are attained and more refinement steps are necessary. However, along with the memory savings, more abstracted variables lead to much faster solve times per error. For instance, b14 requires 3740 second per error with 40% state abstraction while it requires only 172 seconds per error with 82% state abstraction.

It is interesting to notice the relatively small number of iterations necessary to find the injected error. More precisely, b14 and ac97 require only two and three refinement steps, respectively, before finding the errors. This small number of steps indicates that the appropriate variables are selected for refinement and that the debugger is guided efficiently towards the errors after each step.

Overall, the proposed abstraction and refinement debugging framework demonstrates its effectiveness for large problems where conventional approaches may fail due to excessive memory and/or run-time requirements.

V. CONCLUSION

In this work, a novel debugging framework is proposed based on abstraction and refinement. The framework creates an abstract model which undergoes debug and demonstrates substantially reduced memory requirements. By formulating the problem carefully, the overall approach guarantees completeness and correctness. The experiments demonstrate that problems too large for a conventional approach are solved by the proposed framework. Furthermore, memory reductions of over 60% and run time improvements of over 4.5X are observed. Overall, the results encourage further work in the area of abstraction and refinement as an efficient platform for design debugging.

REFERENCES

[1] P. Rashinkar, P. Paterson, and L. Singh, *System-on-a-chip Verification: Methodology and Techniques*. Kluwer Academic Publisher, 1996.

[2] R. Drechsler, *Formal Verification of Circuits*. Kluwer Academic Publishers, 2000.

[3] Y. Yang, S. Sinha, A. Veneris, and R. Brayton, "Automating Logic Rectification by Approximate SPFDs," in *ASP Design Automation Conf.*, 2007.

[4] D. Appenzeller and A. Kuehlmann, "Formal verification of a PowerPC microprocessor," in *Int'l Conf. on Comp. Design*, 1995, pp. 79–84.

[5] M. Abramovici, M. Breuer, and A. Friedman, *Digital Systems Testing and Testable Design*. Computer Science Press, 1990.

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

[7] 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.

[8] 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.

[9] Y. Chen and F. Chen, "Algorithms for compacting error traces," in *ASP Design Automation Conf.*, 2003, pp. 99–103.

[10] K. Chang, V. Bertacco, and I. Markov, "Simulation-based bug trace minimization with BMC-based refinement," in *Int'l Conf. on CAD*, 2005, pp. 1045–1051.

[11] E. Clarke, O. Grumberg, and D. Long, "Model checking and abstraction," in *Symposium on Principles of Programming Languages*, 1992, pp. 342–354.

[12] E. Clarke, A. Gupta, and O. Strichman, "SAT-based counterexample-guided abstraction refinement," *IEEE Trans. on CAD*, vol. 22, no. 7, pp. 1113–1123, 2004.

[13] P. Bjesse and J. Kukula, "Using counter example guided abstraction refinement to find complex bugs," in *Design, Automation and Test in Europe*, 2004, pp. 156–161.

[14] OpenCores.org, "http://www.opencores.org," 2006.

[15] T. Cormen, C. Leierson, and R. Rivest, *Introduction to Algorithms*. MIT Press, McGraw-Hill Book Company, 1990.