Automated Design Debugging With Abstraction and Refinement

Sean Safarpour, *Student Member, IEEE*, and Andreas Veneris, *Senior Member, IEEE*

***Abstract*—Design debugging is one of the major remaining manual processes in the semiconductor design cycle. Despite recent advances in the area of automated design debugging, more effort is required to cope with the size and complexity of today's designs. This paper introduces an abstraction and refinement methodology to enable current debuggers to operate on designs that are orders of magnitude larger than otherwise possible. Two abstraction techniques are developed with the goals of improving debugger performance for different circuit structures: State abstraction is aimed at reducing the problem size for circuits consisting purely of primitive gates, while function abstraction focuses on designs that also contain modular and hierarchical information. In both methods, after an initial abstracted model is created, the problem can be solved by an existing automated debugger. If an error site is abstracted, refinement is necessary to reintroduce some of the abstracted components back into the design. This paper also presents the underlying theory to guarantee correctness and completeness of a debugging tool that operates using the proposed methodology. Empirical results demonstrate improvements in run time and memory capacity of two orders of magnitude over a state-of-the-art debugger on a wide range of benchmark and industrial designs.**

***Index Terms*—Abstraction, debugging, diagnosis, refinement, verification, very large scale integration.**

I. Introduction

THE RELENTLESS consumer demand for devices with complex functionality and superior performance continues to drive the fast growth of the semiconductor industry. As design size gets larger, verification costs also increase significantly. This dramatic increase is confirmed by the number of verification engineers that has quadrupled in a typical chip design team over the last decade [1]. Despite all efforts, detecting errors remains a challenging process as design blocks become larger and the communication between them becomes more elaborate. Once an erroneous behavior is observed, locating the error source is an arduous task in itself since few automated debugging solutions exist to assist the engineers.

Motivated by these needs, functional verification has received much attention from both the academic and industrial communities. An abundance of methodologies have been developed over the past two decades to tackle this challenge. For example, equivalence and property checkers help discover the presence of errors [2], assertion-based verification methodologies and functional coverage tools target functional corner cases [3], and powerful engines such as binary decision diagrams (BDDs), Boolean satisfiability (SAT), and quantified Boolean formula SAT (QBF) solvers extend the capability and applicability of existing computer-aided-design (CAD) verification tools [4]–[7].

These advances target almost exclusively one aspect of verification—that of identifying the presence of errors. Relatively little attention is paid to the task of automating debugging after verification fails. In our context, debugging is the process of locating the source of failure, with the correction task being left to the engineer. Today, debugging is predominantly performed manually due to the scarcity of automated tools. The process is comprised of manually collecting information from the failed simulation trace (counterexample) and manually backtracing from the erroneous signals using "what-if" analysis. This is repeated until the error source is identified. As typical design block sizes today exceed the half-million-synthesized-gates mark and traces range from a few hundreds to a few thousands of clock cycles, it is obvious that scalable automated techniques have become an urgent necessity to alleviate the manual debugging pain and improve the design flow [1].

Traditionally, debugging tools have relied on fault-diagnosis algorithms based on simulation, path tracing, and BDDs [8], [9]. More recently, a new genre of automated debugging methodologies has gained a competitive advantage. These techniques formulate the debugging problem into a SAT instance where a conventional SAT solver can be utilized. Experiments have shown that SAT-based techniques outperform traditional techniques in register transfer level (RTL) debugging [10]–[12]. Parallel to these developments, the use of design hierarchy has helped increase the applicability of existing debugging tools [7]. Despite these significant contributions, automated debugging techniques still find it challenging to cope with today's increasing design size and problem complexity.

Broadly speaking, there are two factors that impact the effectiveness of automated debugging. The first factor is the design size that impacts the solution space. As the gate count increases, the number of suspect error locations that need to be examined also increases, which can dramatically slow down a debugger's performance [10], [12]. Second, the length of the error trace, i.e., the number of clock cycles from the beginning of simulation until the design fails, increases the solution space

Manuscript received June 30, 2008; revised January 13, 2009 and June 18, 2009. Current version published September 18, 2009. This paper was recommended by Associate Editor W. Kunz.

S. Safarpour is with Vennsa Technologies, Inc., Toronto, ON M5V 3B1, Canada (e-mail: sean@vennsa.com).

A. Veneris is with the Department of Electrical and Computer Engineering and the Department of Computer Science, University of Toronto, Toronto, ON M5S 3G4, Canada (e-mail: veneris@eecg.toronto.edu).

Digital Object Identifier 10.1109/TCAD.2009.2030593

that one must examine. Modern debugging solutions must cope with the complexity introduced by these factors to remain practical.

This paper aims to bridge the gap between current debugging capabilities and contemporary industrial needs. It does so by introducing the concepts of abstraction and refinement in automated debugging. In recent years, similar techniques have had a dramatic impact in the scalability and applicability of modern verification methodologies [13]–[15]. Essentially, they simplify the problem by approximating it and iteratively refining it until a solution is found.

In detail, we present the first abstraction and refinement methodology aimed specifically at the automated debugging problem. The proposed methodology starts by creating an initial abstraction model by "simplifying" or removing components of the design. The more components are abstracted, the smaller the problem size becomes, providing a tradeoff between performance and resolution. Next, a conventional debugger is employed to return error locations. If it fails to find any locations due to the level of abstraction, refinement is performed. In essence, refinement systematically reintroduces the abstracted components until the error source is located.

This pairing sequence of debugging and refinement steps is iterated until all solutions are found. A set of theorems, presented in this paper, guarantee the correctness and completeness of this iterative approach. It should be noted that the proposed methodology is not tied to any particular debugging practice. Although the presentation here is conveniently outlined in terms of a SAT-based framework, other diagnosis methodologies (simulation and BDD based) can benefit from the theory developed here as well.

In further detail, this paper introduces a novel abstraction and refinement debugging methodology with the following two abstraction techniques aimed at different debugging applications.

1) *State abstraction*: Aimed at flat netlists, abstraction is carried on state and memory elements to reduce the spatial and temporal problem sizes [16].
2) *Function abstraction*: Leveraging the modular and hierarchical structures of RTL designs, abstraction is performed on functions and modules iteratively in the design hierarchy to reduce the computational requirements.

Extensive experiments on large sequential industrial problems demonstrate memory and run-time reductions of 60% and 4.5× using state abstraction, respectively. With function abstraction, a drastic memory reduction of over 27× and runtime reductions of over two orders of magnitude are observed. This paper clearly demonstrates that abstraction and refinement have a critical impact on the performance of debugging.

This paper is organized as follows. In the next section, notation and background material will be presented. Section III presents the general abstraction and refinement methodology and guarantees its correctness and completeness. Sections IV and V present the details of state- and function-based abstraction techniques, respectively. Empirical results are presented in Section VI, and conclusion and future work are discussed in Section VII.

II. Preliminaries

At the RTL level, a circuit C (combinational or sequential) can be hierarchically composed of modules or functions. In this paper, a function is said to generate a Boolean value for a variable y based on m input variables $x_1, x_2, \ldots, x_m$ and zero or more state variables. For abstraction and refinement, we are primarily concerned with the structural connectivity between the input variables and the variable y of a function. As a result, we label the function of y as $f(x_1, x_2, \ldots, x_m)$ and omit its dependence on any state variables. The terms modules, components, and functions are used interchangeably to refer to entities implementing functions, as defined previously. For example, a Verilog function or a collection of logic gates and flip-flops can define a module. Each module implements a multi-output function $F = \{f_1(X), f_2(X), \ldots, f_p(X)\}$, where each single-output function f_i is defined on input variables $X = \{x_1, x_2, \ldots, x_q\}$. In the remaining paper, single- and multiple-output functions are not distinguished, unless explicitly stated otherwise.

Modules can also contain submodules, thus resulting in a hierarchy tree H for the design [7]. A hierarchy tree H contains nodes representing modules and edges representing parent and child (submodule) relationships. The hierarchy tree H can contain many levels, and each function is tagged with a superscript that indicates its level and a subscript to uniquely label the function. For example, a function F_j^i is at level i of the tree, and it can have subfunctions F_k^{i+1} and F_l^{i+1} at the next level $i+1$. The output of the entire design C is represented by F_1^0 at *root* level 0. This terminology is used extensively in Section V when introducing hierarchical abstraction.

A. Debugging Background

This paper is concerned with functional design debugging, while other types of failures, such as timing and power, are not examined. Furthermore, debugging of test benches or design specifications is outside the scope of this paper. When a failure occurs, functional verification tools such as simulators and property checkers provide traces or counterexamples to reproduce the failure. Although there can be many errors in a design, verification engineers typically focus on a single failure at a time (i.e., the first occurrence of an error at a single observation point). This is because test benches and formal engines generate traces that target a specific functionality or property of the design. Dealing with one failure at a time reduces the probability of multiple errors interacting and often allows the engineer to isolate different error sources.

In this paper, a *diagnosis vector* v is a composed of three sets. One set contains the sequence of primary input logic values needed to simulate the failure. The second set contains the initial state values for all state elements. The third set contains the sequence of correct (reference) values at the observable signals, given the primary input-value sequence. When given multiple diagnosis vectors, the resulting set is labeled as V. An important assumption is that the reference golden model acts as a "black box" to the algorithm. In other words, it can only be simulated to provide the correct output values for the vectors

V. For instance, the golden reference can be written in some high-level language (C/C++, Matlab, etc.), and when contrasted to the RTL, usually expressed in some different hardware description language (HDL) such as Verilog or VHDL, it can provide no structural similarity.

Given an erroneous design C with a corresponding set of diagnosis vectors V that detect a failure, *automated design debugging* is a process that returns suspect components in the design. We say that a component with output function f is a *suspect* if and only if there exists a new function g that can replace f and remove the failure for the set of diagnosis vectors V. Similar to fault equivalence, error suspects can be functionally *equivalent* under V if they cannot be functionally distinguished from one another [8]. Intuitively, functions cannot be distinguished if they produce the same output values, given the same input vectors. In general, when more vectors are available, fewer equivalent suspects exist, leading to a better resolution/accuracy. For a debugging methodology to remain complete and return the actual error site, all equivalent error locations must be returned by the tool.

It has been shown that the complexity of the debugging problem increases exponentially with respect to the number of errors sought or *error cardinality* N. In this paper, a user-defined number $maxN$ denotes the maximum number of errors that the debugger is limited to find. If $maxN$ is smaller than the actual number of errors, then not all error locations can be found by the debugger. During operation, a debugger begins with $N = 1$ and increases N until $maxN$ is reached.

B. SAT-Based Debugging

The first SAT-based debugging formulation is introduced in [10], and it is enhanced to tackle hierarchical design problems in [7]. Another work formulates the problem as a QBF instance where universal quantifiers reduce the memory footprint [7], [17]. Techniques using maximum satisfiability (Max-SAT) and UNSAT core analyses [11], [18] are most effective as quick preprocessing screening steps to drastically reduce the number of components to consider during debugging.

In SAT-based debugging, the problem is represented as a Boolean SAT instance where any SAT solver can be utilized to return solutions corresponding to error locations. The basic idea is to introduce extra hardware to constrain the problem. The four central steps in this process are as follows [10].

1) Add extra logic to the erroneous circuit C to model potential error suspects and represent the error cardinality.
2) Convert it into conjunctive normal form (CNF).
3) Replicate and constrain the CNF for every failing vector sequence in $v \in V$ and for every time frame of v.
4) The final CNF is given to any SAT solver to find the error suspects for vectors V.

We now explain these four steps in terms of a combinational circuit since they are relevant to work here. In the following, we do not distinguish between circuit elements and their corresponding CNF variables. The interested reader should refer to [7], [10], and [17] for more details.

A *correction model*, represented by a multiplexer m_i, is added at the output of every gate (or module) l_i and input

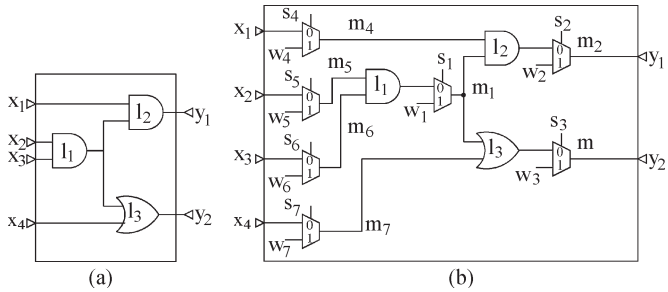

Fig. 1. Circuit before and after adding correction models.

of the circuit C, as shown in Fig. 1. The output of each multiplexer m_i is connected to the original fan-out of l_i. In effect, when the select line s_i of m_i 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 introduced. Eventually, the CNF variable corresponding to w_i will assume a value that corrects the circuit for a single input vector v if l_i is a suspect. In practice, correction models are only applied to gates/modules that are suspicious via cone-of-influence [2], Max-SAT [11], UNSAT core [18], or hierarchical [7] preprocessing to improve performance while searching for solutions.

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 each diagnosis vector v. These constraints are unit clauses corresponding to the Boolean values encapsulated in v. To constrain the SAT problem to a particular error cardinality N, further logic is added to activate at most N select lines. Thus, for $N = 1$, a single s_i can be set to logic 1 at the time, which, in turn, indicates that l_i is an error suspect. For higher values of N, the number of multiplexer select lines set to logic 1 indicates an N-tuple error suspect. An all-solution SAT solver can be implemented to return all equivalent error suspects, as described in [10]. Iterations of all-solution problems can be formulated from $N = 1$ to $N = maxN$ to locate all possible errors with at most cardinality $maxN$.

C. Abstraction and Refinement in Model Checking

Abstraction and refinement techniques are used readily in model checking to mitigate the exponential nature of the underlying state space [13]–[15], [19]. Roughly speaking, an *abstract model* is derived by removing state elements or other components from the original *concrete* design. As an active area of research, many different types of abstraction techniques exist, such as existential abstraction and predicate abstraction [20]–[22]. Irrespective of the abstraction approach, the final abstract model contains fewer circuit elements than the original one, thus simplifying the task of the model checker.

Depending on the properties being verified and the abstraction technique used, the model checking result may or may not be trusted. For example, consider the scenario when verifying a universal property (whether the property holds for all paths) using an existential-abstraction technique [23]. If model checking determines that a property holds in the abstract model, then it must also hold in the concrete design [23]. However, if a property does not hold in the abstract model, then the corresponding counterexample must be validated in

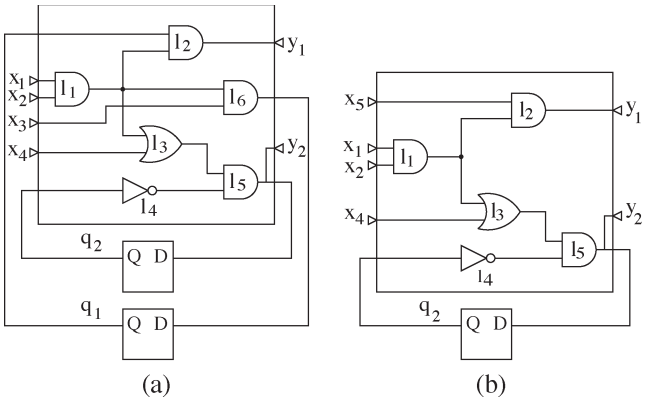

Fig. 2. Circuit before and after abstracting flip-flop q_1.

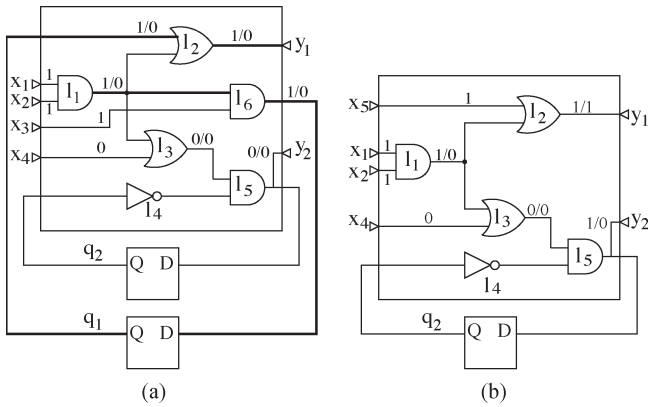

Fig. 3. Demonstrating the effect of unconstrained inputs on the abstract circuit.

the concrete design. If the counterexample does not expose a failure of the property in the concrete design, then it is said to be *spurious* [21]. In this case, the abstract model is refined by reverting some of the abstracted components and continuing the model-checking process.

III. Debugging With Abstraction and Refinement

The aim of abstraction-based debugging is to reduce the size and complexity of the underlying problem. Since the performance of a debugger and its memory requirements are directly related to the size of the circuit under analysis, abstraction can introduce considerable run-time and memory benefits. This section introduces the basics of a complete and sound debugging methodology using abstraction and refinement.

An abstract model C' is derived by removing a set of components or functions Abs from a concrete model C. More precisely, as shown hereinafter, the procedure $selAbsComponents$ selects a set of functions to abstract, while the procedure $absDesign$ removes these from design C

$$Abs = selAbsComponents(C)$$
$$C' = absDesign(Abs, C).$$

When the components Abs are removed, some of the circuitry in their transitive fan-in may be left dangling (i.e., unused by any other logic). An iterative dangling logic removal procedure can eliminate all gates, wires, and other state elements unused by the abstracted components [24]. The resulting model C' can be significantly smaller than C. For instance, if Abs includes all primary outputs of a circuit, the entire circuit can be essentially removed. The degree of abstraction to perform is addressed through the experiments of Section VI.

After removing the components Abs, their direct fan-outs, which are now undriven, are connected to newly introduced primary inputs. Specifically, for every function $f_i \in Abs$, a new primary input is introduced in C' and connected to the fan-out of f_i. As an example, consider Fig. 2(a) and (b), where a circuit is shown before and after abstraction, respectively. In this case, the component to abstract is $Abs = \{q_1\}$. Notice that q_1 and its transitive fan-in logic, l_6, and x_3 are removed in Fig. 2(b) and that the fanout of q_1 (i.e. l_2) is now driven by the new primary input x_5.

A. Guaranteeing Correctness

Once the abstract model C' is generated, the next step is to construct the debugging problem. The first attempt is to formulate the problem according to Section II-B with the abstract model C' and the error trace V. However, the abstract model C' contains the newly added primary inputs, which remain unconstrained in V. As a result, a SAT-based debugging engine may arbitrarily assign *unjustifiable* logic values to these variables while solving the debugging problem.

Definition 1: Assume that $\Phi(\Phi')$ corresponds to a SAT-based debugging problem derived from a concrete design C (abstract design C'). A value assignment to the variables of Φ' is *unjustifiable* if the same assignment to Φ gives a conflict.

Here, a conflict occurs when there is an attempt to assign different values from the Boolean domain to the same variable. Consequently, the solutions returned by a debugger in this formulation cannot be trusted because it may be incorrect. The following example illustrates this particular scenario.

Example 1: Fig. 3(a) shows a concrete design with an error on gate l_1. Regardless of the error type, the correct/erroneous logic values 1/0, shown in bold, propagate from gate l_1 through flip-flop q_1 and to primary output y_1. Notice that the primary input values remain constant in 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 a value 1, which will produce the correct/erroneous value pair 1/1, as shown in Fig. 3(b). Here, the value assignment of $x_5 = 1$ is unjustifiable because, in the concrete design of Fig. 3(a), the corresponding assignment to q_1 is 0.

One way to prohibit unjustifiable solutions from occurring is to constrain the newly added primary inputs to the values of the Abs components in design C, as proposed by Theorem 1.

Theorem 1: Given a circuit C, an input vector sequence from v, the set Q contains the simulation values of the output of components Abs for all clock cycles in v. A debugging problem formulated with abstract model C' and $v' = v \cup Q$ will not have any unjustifiable assignments.

Proof: This proof is based on the fact that the abstract model can be restricted sequentially to behave like the concrete model. For every clock cycle, the fan-out logic of every Abs component in C is driven by circuit elements whose Boolean

values are stored in Q. Similarly, the Boolean values in Q are used to drive the new primary inputs in C' for every clock cycle. Since the fan-out logic of every Abs component in C' is constrained to the same values as in C, unjustifiable assignments will not occur. ■

By Theorem 1, *correctness* is guaranteed since all solutions for the abstract model correspond to solutions for the concrete design. Next, the proposed methodology is extended to find suspects that may be accidentally abstracted.

B. Spurious Solutions

Since abstraction may remove large sections of a design, it is possible that error sources are accidentally removed. This case will be identified, as automated debuggers will return the new primary input suspects as spurious solutions.

Definition 2: Spurious solutions are primary input suspects returned by an automated debugger that correspond to the abstracted components.

Spurious solutions do not provide enough information about the error source to help rectify the erroneous concrete design. In other words, these spurious solutions mask equivalent error locations. To find the error locations in the concrete design, the abstracted variables and their respective removed fan-in logic must be analyzed. One way is to refine the design based on the spurious solutions and iterate the debugging process. Refinement is achieved by reintroducing the original circuitry, including the removed fan-in logic, corresponding to the spurious solutions into the design C'.

Example 2: Consider the circuit in Fig. 3(a) after abstracting q_1, where a debugger finds l_2, l_1, and x_5 as suspects with $N = 1$. Here, location l_6, which is the error source in the concrete design, is abstracted. In this case, the spurious solution x_5 masks the error source l_6. Refinement is necessary to reintroduce l_6 into C', thus allowing the debugger to find l_6 in the next iteration.

Traditionally, complete solutions that return all equivalent solutions are important in debugging [10] since they offer more degrees of flexibility for the designer to correct the design or optimize it, if a debugging-based rewiring algorithm is used [25]. To find all equivalent suspects, all solutions corresponding to the abstracted components must be refined, a process performed iteratively until no more solutions from the abstracted components are found. In practice, since the proposed process is incremental, the user, at any time, can attempt to rectify the circuit before the entire debugging process is complete.

C. Guaranteeing Completeness

The abstraction formulation and refinement schemes discussed in the previous sections provide a means of identifying error sources without considering the entire design. However, under certain conditions, some equivalent solutions may be missed by the debuggers. This happens when a set of m errors in the concrete design are mapped onto a set of n errors in the abstract model, where $n > m$, as shown in Example 3.

Example 3: Consider the abstract circuit in Fig. 2(b) unfolded for two time frames, as shown in Fig. 4. For clarity, the abstracted logic l_6 is shown in dashed lines. Notice that

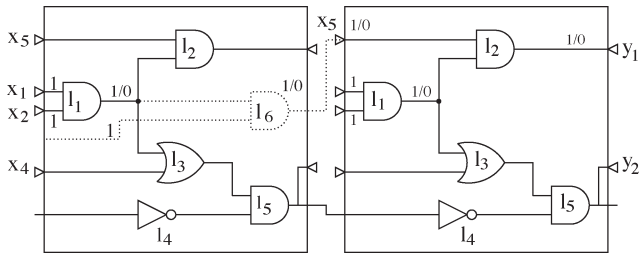

Fig. 4. Abstract circuit unfolded over two time frames.

the error from gate l_1 does not directly propagate to output y_1, but its effect is captured in abstract variable x_5. For error cardinality $N = 1$, the SAT solver returns the single equivalent error location l_2. Assuming that the design is analyzed and that 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 $\{l_1, x_5\}$ 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.

The aforementioned example illustrates how abstraction can cause the error location to be found with higher error cardinality. Given a maximum user-defined error cardinality of $maxN$, when using abstraction and refinement, the maximum cardinality should be set to $maxN_{abs} = maxN + |output(Abs)|$, where $|output(Abs)|$ is the number of outputs for the abstracted functions (or the number of new primary inputs). Theorem 2 presents the steps required to find all equivalent error locations for a user-specified value of $maxN$.

Theorem 2: Assume that a debugger returns solution set S for concrete design C, diagnosis vectors V, and maximum error cardinality $maxN$. The debugging procedure that performs the following steps with an abstract model C', diagnosis vectors V', and maximum error cardinality $maxN_{abs} = maxN + |output(Abs)|$ finds set of solutions $S' \supseteq S$.

1) Initialize N to 1.
2) Debug C' with V' and N to get solution set S'.
3) If any solutions $s \in S'$ are spurious, refine the abstract model C' using s, and then go to step 1).
4) Increment N by 1.
5) If $N > maxN_{abs}$, return S'; else, go to step 1).

Proof: In the worst case, some error sources are abstracted, and their behavior is captured by the new primary inputs or $output(Abs)$. Together, the maximum number of active error locations is $maxN_{\text{abs}} = maxN + |output(Abs)|$. The debugger proceeds to find solutions based on the abstract model using $N \leq maxN_{\text{abs}}$. If any of the solutions are spurious, then the abstract model is refined, and those variables are replaced with their corresponding concrete components. The new abstract model is then given to the tool that starts the search with $N = 1$ again. The search continues until $N = maxN_{\text{abs}}$, and all the equivalent errors that map into $maxN_{\text{abs}}$-tuples or fewer will be found. After every refinement step, some abstracted components are reintroduced, and previous solutions at $N = maxN_{\text{abs}}$ may be found at $N \leq maxN_{\text{abs}}$. This process guarantees that all the abstracted components that mask error locations are systematically resolved, thus finding all the solutions in S. ■

Algorithm 1 Debugging With Abstraction and Refinement
1: $S = \emptyset$, $N = 1$
2: $Abs = selAbsComponents(C)$
3: $C' = absDesign(Abs, C)$
4: $maxN_{\text{abs}} = maxN + |outputs(Abs)|$
5: **while** (1) **do**
6: $V' = \text{extract\_constraint}(C, C', V)$
7: $New\_sols = debug(C', V', N)$
8: **for all** $Sol \in New\_sols$ **do**
9: **if** (spurious_solutions (Sol, C')) **then**
10: $C' = \text{refine}(Sol, C', C)$
11: $N = 0$
12: $maxN_{\text{abs}} = maxN + |outputs(C', C)|$
13: **else**
14: $S = S \cup Sol$
15: **end if**
16: **end for**
17: $N = N + 1$
18: **if** $(N > maxN_{\text{abs}})$ **then**
19: return $\{S, C'\}$
20: **end if**
21: **end while**

D. Overall Algorithm

Algorithm 1 illustrates the overall abstraction and refinement scheme for a debugging methodology that guarantees correctness and completeness. The first step is to generate the initial abstract model C', as shown in lines 2 and 3. To ensure correctness, in line 6, the stimulus is modified to constrain the new primary inputs to their simulation values, as discussed in Section III-A. The modified diagnosis vector V' and abstract model C' are provided to the debugger to find the error locations, as shown in line 7. Next, according to the spurious solutions, refinement may be performed, the error cardinality is reset, and $maxN_{\text{abs}}$ is recalculated. If the solutions are not spurious, then they are added to solution set S to be returned to the user. The aforementioned steps are repeated until $maxN_{\text{abs}}$ is reached for completeness.

Even though the final solution S is returned in line 19, the algorithm is incremental in nature, meaning that every solution found can be provided to the user. The benefit of an incremental algorithm is that suspects can be analyzed by engineers prior to all equivalent solutions being found.

IV. State Abstraction

State abstraction is one type of abstraction where memory elements such as flip-flops and latches are selected for removal. This approach can be powerful because state elements are important components that play a central role in both state machine and datapath logic. Furthermore, when modular or hierarchical information is not available for a design, as is the case for post-synthesis netlist or custom logic, state abstraction can operate on the flat design.

The effectiveness of state abstraction is demonstrated empirically in Section VI-A. A subtle benefit of state abstraction is that, with a reduced state space, debug traces can be considerably shortened. This advantage will be discussed next.

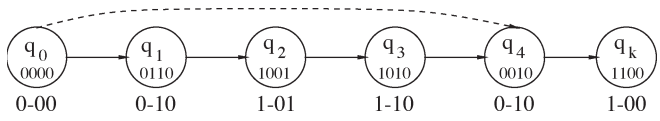

Fig. 5. Reduced trace V' due to abstraction.

A. Trace-Length-Reduction Benefits

Long error trace lengths are commonly associated with simulation-based verification tools where random and constrained-random stimuli are used to exercise the design. Both manual and automated debugging can benefit from operating on shorter error traces. Trace reduction is an effective preprocess to debugging because it can reduce trace lengths by orders of magnitude [26]–[28].

State abstraction can help further reduce the trace length prior to debugging. With many of the state elements being abstracted, the state space of the design is reduced, thus allowing for state-matching techniques to remove repeated states and redundant transitions [26]–[28]. It should be emphasized that most state-matching techniques implicitly resimulate reduced traces in order to ensure that the desired failure is still exposed.

As an example, consider Fig. 5, where a state-transition diagram is used to illustrate an error trace from states 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), states q_1 and q_4 can no longer be differentiated. The state values after abstraction are shown under each node in Fig. 5. As a result, a shortcut can be taken in the trace from states q_0 to q_4, as illustrated by the dashed line. Note that, as required by most trace-reduction techniques, the compacted traces must be tested on the concrete design to determine whether the error(s) are/is still observable.

V. Function Abstraction

When a design contains high-level or RTL information, function abstraction can provide a natural and powerful way to partition the debugging problem. For instance, designers working on RTL designs use modules to partition the design based on functionality and complexity. These modules are also good candidates for abstraction. Furthermore, the hierarchical and modular compositions of HDL designs can be leveraged to apply abstraction and refinement in a systematic manner.

A. Hierarchical Abstraction

The strength of function or modular abstraction can be amplified when used in a hierarchical manner. More specifically, module-based debugging can be applied iteratively at each hierarchy level, thus allowing for a divide-and-conquer debugging approach.

At each level i of hierarchy H, the functions $\{F_1^i, F_2^i, \ldots, F_p^i\}$ can be considered by the procedure $selAbsComponents$ to select the components to abstract

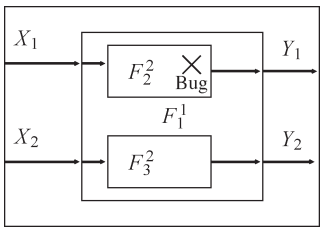

Fig. 6. Function F_1^1 is composed of functions F_2^2 and F_3^2.

(Abs_i). The iterative sequence of abstraction, debugging, and refinement presented in Algorithm 1 can be applied to the problem constructed at hierarchy level i. However, only functions at level i can be refined and not their subfunctions. In order to locate the errors in the subfunctions, the entire algorithm must be repeated at hierarchy level $i+1$.

Two properties of hierarchical abstraction and refinement are very important. After completing an iteration of Algorithm 1 at hierarchical level i, the following properties hold true.

1) If a function f^i is still abstracted, then its subfunctions at g^j can be abstracted at hierarchy levels $> i$.
2) If a function f^i is refined, then its subfunctions at g^j may still be abstracted at hierarchy levels $> i$.

The first observation is easy to confirm. When a function is still abstracted after debugging, it signifies that equivalent error locations do not reside inside it. Similarly, the subfunctions will not contain any equivalent error locations either, and they should be abstracted at deeper hierarchy levels.

For the second observation, consider Fig. 6, where an error resides in F_2^2. At level 1, function F_1^1 cannot be abstracted since it contains an error. However, at level 2, subfunction F_3^2 may be abstracted since it is independent from F_2^2 and its output. Thus, functions can be partitioned into subfunctions such that some of the subfunctions will not contain any equivalent error locations.

B. Overall Algorithm

To reduce the debugging problem size further, when operating at a given hierarchy level i, all functions at a deeper hierarchy level $> i$ should also be abstracted in C'. However, it is important to only refine modules at level i. This restriction reduces the complexity of the debugging problem at level i and postpones the analysis of the subfunctions at level $> i$ to future hierarchy levels.

The process of finding all equivalent solutions through the management of error cardinality is the same with hierarchical abstraction from Section III, as shown in the following.

Example 4: Consider Fig. 7(a), where the modules $Abs_1 = \{F_2^1, F_4^1\}$ are abstracted at level 1. The abstraction results in the removal of modules F_1^1 and F_3^1 as well because they fan-in to Abs_1. The initial abstracted circuit is shown in Fig. 7(b). Assuming that the error is in module F_7^2, the error effect can propagate to the output of Y_1 and Y_2. In this example, the debugger will not identify a single error source but will find the error pair of $\{X_{F_2^1}, X_{F_4^1}\}$ with $N = 2$. Through refinement, these modules and their fan-in circuitry are reintroduced in the

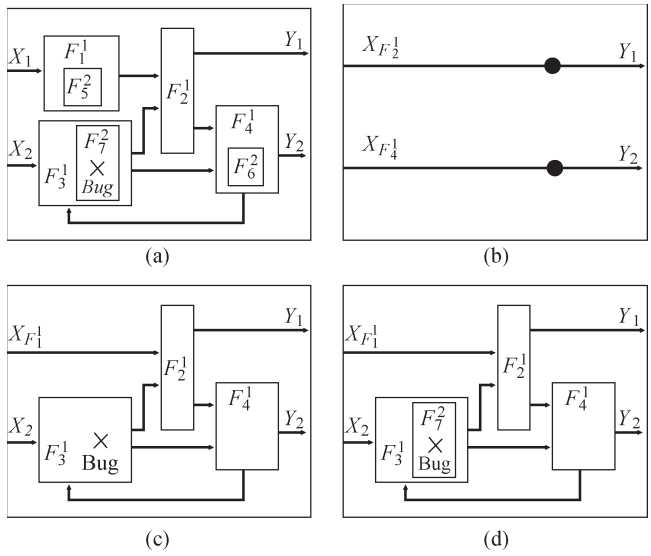

Fig. 7. Hierarchical abstraction and refinement example. (a) Model C before abstraction. (b) Initial model C' at level 1. (c) Final model C' at level 1. (d) Final model C' at level 2.

circuit, as shown in Fig. 7(c). Next, the error cardinality N must be reset to 1. At hierarchy level 2, the modules F_7^2 and F_6^2 can be abstracted as part of Abs_2. Refinement will reintroduce module F_7^2, and debugging will find the error source inside it, as shown in Fig. 7(d).

The proposed hierarchical abstraction and refinement methodology is shown in Algorithm 2. Here, the debugging problem is solved iteratively by descending hierarchy H. At each hierarchy level i, procedure $absDesign$ first abstracts all functions at levels $> i$. This ensures that subfunctions will not be refined. Next, function abstraction and refinement is performed by *Function_debug* according to Algorithm 1. The effectiveness of the proposed technique is demonstrated in the experiments of Section VI.

Algorithm 2 Hierarchical Debugging
1: $Solutions = \emptyset$, $level = 0$, $N = 1$, $C' = C$
2: **while** (1) **do**
3: $level = level + 1$
4: $C' = absDesign(level + 1, C')$
5: $\{New\_sols, C'\}$ = Function_debug$(C', level, N)$
6: **if** $New\_sols = \emptyset$ **then**
7: return *Solutions*
8: **else**
9: $Solutions = Solutions \cup New\_sols$
10: **end if**
11: **end while**

VI. Experiments

This section evaluates the effectiveness of the proposed abstraction and refinement debugging methodology. First, state abstraction is applied to gate-level diagnosis problems. The second set of experiments is conducted on RTL designs that are developed in a hierarchical manner. For those circuits, function and hierarchical abstraction and refinement are used.

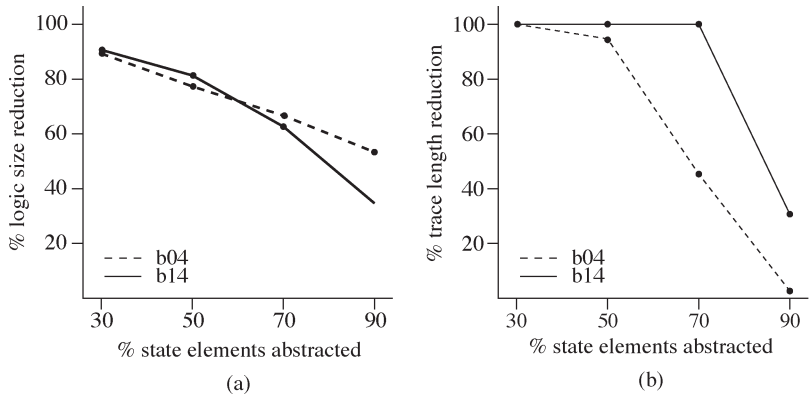

Fig. 8. Logic and trace reduction versus flip-flops abstracted.

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	-	-	-

A. State Abstraction

To evaluate the effectiveness of state abstraction, hand-made bugs are inserted in circuits from the ISCAS'89 and ITC'99 benchmarks, as well as industrial RTL circuits from www.OpenCores.org [29]. The bugs are single-gate or RTL assignment changes following Abadir's model [30]. As discussed in Section II-A, diagnosing single errors is very common in practice, as traces usually target specific design functionalities. The single errors inserted are also adequate models for applications such as post-synthesis netlist and custom logic diagnosis [8], [27], electronic change order (ECO) modifications [31] and automated-correction applications [32].

For each erroneous circuit, ten traces that fail are obtained through pseudorandom simulation. The debugger used to locate the error sites is the SAT-based methodology proposed in [10], with MiniSAT [33] being the underlying SAT solver. The suspects found using all ten traces are intersected to return the final suspect set. In the proposed abstraction and refinement procedures of Section IV, the design and traces are modified from C and V to C' and V', respectively, before the debugging engine is called. Contrasting the performance of the debugger before and after applying the proposed techniques provides a fair-comparison metric.

The experiments are conducted on a 2.66-GHz Intel Xeon processor with 2 GB of memory and a time-out of 7200 s for each problem. In each case, a basic trace-compaction procedure is applied before debugging. This procedure first builds a graph for the visited states, it then connects edges between repeated states, and it finally applies Dijkstra's shortest path algorithm from the initial state to the final state [34]. More sophisticated trace-compaction schemes may provide better results [26], [27]. The resulting traces that do not distinguish between the reference and buggy circuits are discarded.

In Section IV, the effects of abstraction on logic size and trace length are discussed. Fig. 8 summarizes the relationship between the degree of abstraction performed and its consequence on circuit size and trace length. This general behavior is illustrated using the `b04` and `b14` benchmarks. Fig. 8(a) shows an apparently linear relationship between logic-size reductions and the number of abstracted state elements. In Fig. 8(b), the experiments show that significant trace-length reductions are possible only after a certain threshold is 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 benchmark characteristics and performance statistics when debugging the concrete circuits. Columns 1, 2, and 3 present the circuit name, number of gates, and number of flip-flops (state elements) in each circuit, respectively. Columns *# clk* and *# red. clk* show the average lengths of the traces before and after trace compaction, respectively.

The next five columns summarize the results of the debugger for each problem. In columns *# cls* and *mem (MB)*, the number of clauses (in thousands) generated for each problem and memory usage are presented, respectively. The average time

TABLE II
PERFORMANCE STATISTICS FOR ABSTRACTION AND REFINEMENT DEBUGGING FRAMEWORK

circuits	red. logic(%)	red. FF(%)	red. trace(%)	red. mem(%)	time/err (s)	# err	$maxN_{abs}$	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	**-**

required to find the errors and the number of equivalent errors found by the tool are presented in columns *time/err (s)* and *# err*, respectively. Finally, the total time required to find all the errors is presented in column *total (s)*. To cope with the size of the larger problems, the CNFs are partitioned into bands (groups) for each trace vector, and each is solved sequentially, as described in [10]. For `b14` and `ac97`, where the average reduced traces are 54 and 30 time frames long, respectively, the problems still run out of memory. As we shall see, the proposed abstraction framework is particularly beneficial in such memory-intensive cases.

Table II presents the results when the tool utilizes the proposed abstraction and refinement techniques. For each problem, state elements are randomly selected such that between 40% and 50% are abstracted, which are conservative amounts according to Fig. 8. More powerful abstraction heuristics can be developed to intelligently select which components to abstract; however, this basic scheme adequately demonstrates the effectiveness of our methodology.

To allow a comparison with the data in Table I, the percentage of logic and flip-flops still abstracted after the iterations of Algorithm 1 is presented in columns 2 and 3. These figures also included logic and other state elements removed through dangling logic removal. Additional compacted traces and overall reduced-memory requirements are shown in columns 4 and 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 version. The largest problems in Table I are for circuits `b14` and `ac97`, and they ran out of memory. With the new methodology, they both successfully complete. It can be calculated that, on 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. Furthermore, Fig. 8(b) states that 40%–50% state abstraction does not result in reduced traces for most problems. In the experiments, the initial trace-compaction process is able to reduce the traces considerably. For instance, the initial trace of circuit `s1488` that is 104 clock cycles is reduced to only five clock cycles after compaction, so further reduction is highly unlikely. For industrial traces derived from functional test benches and not randomly, it is highly unlikely to reduce their length drastically by simple state-matching techniques [26]. 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 (s)* 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 is 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 inserted errors), then refinement is not performed. If the errors found by the proposed framework do not include all equivalent error locations (i.e., *# err* is smaller in Table II than in Table I), then all spurious solutions must be refined.

In Table II, column $maxN_{\rm abs}$ shows the maximum number of tuples searched until all equivalent errors are found. The debugging time for all searches prior to $maxN_{\rm abs}$ is shown in column *prev (s)*. When refinements are necessary, column *refine (s)* presents the solve time for the subsequent searches.

For many of the problems in Table II, the maximum error tuple found $(maxN_{\rm abs})$ is often greater than 1 but always less than or equal to 3. This signifies the fact 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 $maxN_{\rm abs}$ (*prev (s)*) is always quite smaller than the average time required to find an error (*time/err (s)*). If we take `b12` for instance, it takes on average 4.2 s to determine that no errors occur when $N < 3$ and 85 s to find each solution at $N = 3$. Relating these times to Algorithm 1, it means that the approach is quite effective since the majority of time is spent in the debugging function in line 7 when $N = maxN_{\rm abs}$ and not when $N < maxN_{\rm abs}$.

The total debugging time for the proposed approach is found by summing the product of *time/err (s)* and *# error* with *prev (s)* and *refine (s)*, and it is shown in column *total (s)*. Performance improvement over that data from Table I is given in column *X impr*. When abstracting 40%–50% of the state elements, not many refinement steps are necessary, as most equivalent error

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

locations are found in the abstract model. However, even for the cases where refinement is necessary, substantial run-time improvement is observed, with the exception of benchmark b12. Overall, performance improvements of up to 4.5× are observed, with an average value of 2× across all problems. This increased efficiency can be attributed to CNF problems of smaller size that are easier to get tackled by the solver.

As observed in Fig. 8, 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 IV, with 80% and 96% of the state elements being abstracted, respectively. For easy comparison, the first row of each table represents the problem properties of Table II. The following rows show the results after each abstraction and refinement step until the specific injected error is found (not all equivalent errors as in Table I). For each table, column 1 describes whether the data are derived from Table 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 savings are attained, yet 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 s/error with 40% state abstraction, while it requires only 172 s/error with 82% state abstraction. It is interesting to notice the relatively small number of iterations that are necessary to find the injected error. More precisely, b14 and ac97 require only two and three refinement steps.

As shown in Tables III and IV, aggressive abstraction schemes can be desirable at times since incremental solutions are quickly returned. However, many refinement steps may be required to find all solutions. For instance, to find all equivalent solutions for b14, the percentage of logic abstracted must reach 15.3% from 50.1%. This will likely require many more iterations of refinement according to the linear curve of Fig. 8(a).

B. Function Abstraction

This section presents the experiments for function abstraction. All the circuits used are from the www.OpenCores.org [29], except for an industrial communication design (*comm*) that has nearly 500 000 synthesized gates. In this set of experiments, hierarchical information about the design is readily available to the methodology.

TABLE V
SUMMARY OF PROBLEMS FOR FUNCTION ABSTRACTION

design	Problem statistics			Debugger engine [7]		
	size	# FF	# clk (used)	# literal	time (s)	mem (M)
wb_con1	80695	818	19 (19)	518580	58.74	619
wb_con3	80695	818	1387 (40)	1273699	205.16	1250
fdct1	264221	5461	189 (40)	1705328	555.37	4400
mem_ctrl1	38660	1145	1318 (40)	3887703	55.13	850
vga1	147457	17102	16100 (40)	8679788	1635.78	4700
vga2	147457	17102	141 (40)	212588	236.16	1350
comm1	449927	30339	19 (25)	1912087	1575.67	5080
comm2	453788	26852	88 (25)	Mem out	Mem out	8000
comm3	453576	26852	1387 (25)	277649	809.31	4831

Each circuit contains functional-level errors, such as incorrect operation, incorrect module instantiation, bad module wiring, wrong state-machine transition, etc. Bugs are inserted systematically into the designs to represent typical human-made RTL-level errors. It is important to notice that these RTL errors usually translate tens or hundreds of errors in the synthesized gate-level netlist. An industrial simulator with a behavioral Verilog test bench accompanying the design is used to identify the presence of the bugs. The debugger utilized is the module-aware SAT-based one from [7] implemented on top of MiniSAT [33]. The experiments are conducted on a 64-b Intel Core 2 Quad processor with 2.66 GHz and 8 GB of memory.

Table V presents a summary of the debugging problems and the corresponding tool statistics. Columns 1, 2, and 3 show the name of the design, and its size in terms of gates and state elements (FF), respectively. Column 4 contains the number of cycles for the failing input trace. When it is too long for the tool to formulate the problem, the trace is reduced to only contain the last 25 or 40 simulation cycles. The final number of clock cycles used to build the problem is given in parentheses in column 4. For example, problem wb_con2 contains 1397 clock cycles, but only the last 40 clock cycles are used. The next column presents the number of literals in the CNF. Finally, columns *time* and *mem* show the total run time (in seconds) to solve the problem and the memory usage (in megabytes). Notice that comm2 requires more than 8000 MB to formulate it, and it runs out of memory.

Table VI presents the result of the proposed technique on the debugging problems. For these experiments, all modules are initially abstracted. Column 1 shows the problem name, while column 2 shows the maximum error cardinality ($maxN_{\mathrm{abs}}$) to solve it. As discussed in Section III-C, the cardinality required to locate the bug using an abstracted design can be larger than the actual number of errors. This is demonstrated in the comm1 and comm3 instances where a higher cardinality of 2 is used to find the single error.

In Table VI, the column labeled *# itr* states the number of refinement and debugging iterations (line 7 in Algorithm 1) to find all the equivalent locations. The column *mod refined/total* presents the number of modules refined out of the total number of modules in the concrete design. These modules are the only

TABLE VI
RESULTS OF THE PROPOSED FUNCTION ABSTRACTION AND REFINEMENT TECHNIQUE

design name	abstracted problem stats						comparison to original		
	$maxN_{abs}$	# itr	mod refined/total	# literals	time (s)	peak mem (M)	lit reduced (×)	speed up (×)	mem reduced (×)
wb_con1	1	3	3 / 8	115547	25.55	253	4.49	2.30	2.45
wb_con2	1	4	4 / 8	140713	149.12	469	9.05	1.38	2.67
fdct1	1	6	5 / 5	1705328	638.78	4400	1.00	0.87	1.00
mem_ctrl1	1	4	12 / 14	112581	12.02	200	34.53	4.59	4.25
vga1	1	2	5 / 14	13767	6.27	173	630.48	260.89	27.17
vga2	1	5	6 / 14	94066	436.38	1052	2.26	0.54	1.28
comm1	2	8	10 / 129	37960	108.32	772	50.37	13.11	6.58
comm2	1	9	10 / 129	25105	1403.47	640	—	—	> 12.50
comm3	2	8	8 / 129	80103	63.94	317	3.47	12.66	15.24

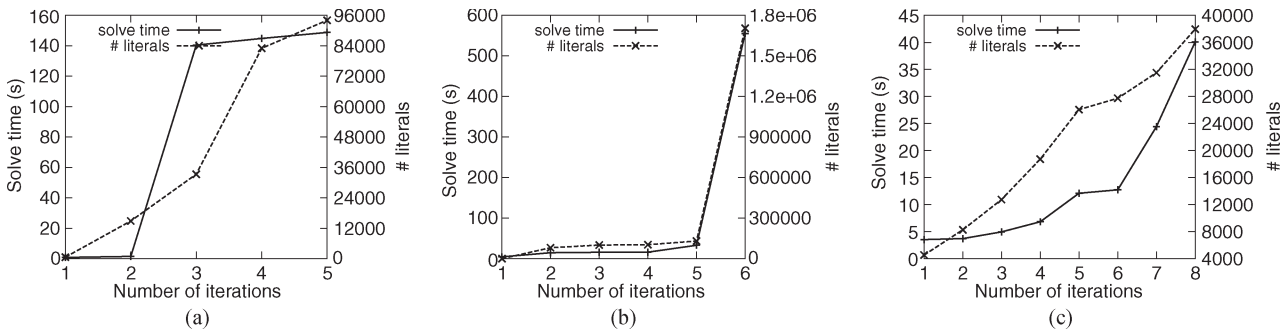

Fig. 9. Solve time and # literals in problem versus the number of refinement and debugging iterations for `vga2`, `fdct1`, and `comm1`.

ones required to diagnose the error. The smaller this number is, the more effective abstraction/refinement becomes. The next three columns, namely, *# literals*, *time (s)*, and *peak mem (M)*, present the benefit of the proposed technique in terms of the number of problem literals, total run time (in seconds), and peak-memory requirements.

The improvement provided by the proposed technique is shown in the last columns of Table VI. Here, the reduction in the number of literals, the speedup in run time, and the reduction in memory over the debugging technique of [7] without abstraction and refinement are shown. The effectiveness of the abstraction technique is attributed to the reduction of problem size or literal count. For example, consider problem `vga1` where 5/14 modules are used, leading to 630.48× reduction in literals, which results in 260.89× improvement in run time and 27.17× reduction in the overall memory requirement. For problem `comm2`, which resulted in memory out without the abstraction technique, only 640 MB of the available 8000 MB is required. For all problems, the number of refinement and debugging iterations performed is larger than one. Therefore, it is clear that each iteration is much easier and faster when abstraction is used, so it is more advantageous to run more iterations on easier problems than fewer iterations on harder ones.

In Table VI, there are two problems that experience a slowdown. It is worthwhile to analyze the reason for this behavior. For problem `fdct1`, six iterations are required to solve the problem, at which stage all five modules are used. Thus, in this case, the extra iterations simply add overhead as the entire circuit is needed in order to solve the problem. Problem `vga2` also experiences a slowdown, but in this case, a 2.26× reduction in memory is observed. Unlike the overall trend, the simpler and faster debugging problems cannot compensate for the extra iterations performed.

Fig. 9(a)–(c) shows the detail into the numbers of Table VI for `vga2`, `fdct1`, and `comm1`, respectively. The figure illustrates the relationship between the run time shown in solid line and the number of literals shown in dashed line against the refinement and debugging iterations. Notice the general trend where both run time and number of literals appear to increase exponentially with the increase in the number of iterations. For the majority of cases where the proposed technique is effective, abstraction allows the problem to be solved with a fraction of its size, thus leading to smaller memory requirements and run times. Considering problem `vga2`, notice that, for iterations 3, 4, and 5, the solve time is quite high, thus not providing any run-time benefit.

The proposed techniques allow for different degrees of abstraction to be applied. In general, aggressive (high-degree) abstraction leads to more debugging and refinement steps. However, due to the simplicity of the design when abstracted aggressively, the initial debugging and refinement iterations are relatively much easy problems and thus quicker to solve. This behavior is observed in Fig. 9, where the initial iterations have a faster run time than the latter ones. It may be possible to find an abstraction heuristic that can balance the number of iterations and the functions abstracted, but this is not a trivial task. Overall, these experiments show that abstracting all RTL functions and modules becomes quite effective.

VII. CONCLUSION

As debugging remains a manual and time-consuming burden in today's chip design cycle, it becomes important to develop automated methodologies to address contemporary verification needs. This paper has presented state/function abstraction and refinement techniques in design debugging to allow larger

designs to be handled faster and with less memory by existing CAD tools. Designs were first abstracted, resulting in smaller debugging problems. To ensure that all the equivalent error locations are found in the original design, a refinement process was performed. Refinement was applied in iterations, thus only reintroducing the necessary components for debugging. A consequence of state abstraction is that the error trace can be further reduced, and so does the problem size. Function abstraction employed in a hierarchical framework allows for a powerful debugging framework. The experiments have demonstrated an order of magnitude improvement in both memory requirements and run time for state abstraction and two orders of magnitude for function abstraction. Admittedly, the results from this paper encourage work in this field in an effort to develop scalable and robust automated debugging methodologies.

REFERENCES

[1] "International Technology Roadmap for Semiconductors," *ITRS 2006 Update*, 2008. [Online]. Available: http://www.itrs.net/

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

[3] H. Foster, A. Krolnik, and D. Lacey, *Assertion-Based Design*. Norwell, MA: Kluwer, 2003.

[4] R. Bryant, "Binary decision diagrams and beyond: Enabling techniques for formal verification," in *Proc. Int. Conf. CAD*, 1995, pp. 236–243.

[5] T. Larrabee, "Test pattern generation using Boolean satisfiability," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 11, no. 1, pp. 4–15, Jan. 1992.

[6] 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*, vol. 1579. Berlin, Germany: Springer-Verlag, 1999, pp. 193–207.

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

[8] M. Abramovici, M. Breuer, and A. Friedman, *Digital Systems Testing and Testable Design*. New York: Comput. Sci. Press, 1990.

[9] S.-Y. Huang, "A fading algorithm for sequential fault diagnosis," in *Proc. 19th IEEE Int. Symp. DFT VLSI Syst.*, 2004, pp. 139–147.

[10] A. Smith, A. Veneris, M. F. Ali, and A. Viglas, "Fault diagnosis and logic debugging using Boolean satisfiability," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 24, no. 10, pp. 1606–1621, Oct. 2005.

[11] S. Safarpour, M. H. Liffiton, H. Mangassarian, A. Veneris, and K. A. Sakallah, "Improved design debugging using maximum satisfiability," in *Proc. Int. Conf. Formal Methods CAD*, 2007, pp. 13–19.

[12] K.-H. Chang, I. Markov, and V. Bertacco, "Automating postsilicon debugging and repair," *Computer*, vol. 41, no. 7, pp. 47–54, Jul. 2008.

[13] E. Clarke, O. Grumberg, and D. Long, "Model checking and abstraction," in *Proc. Symp. Principles Programm. Languages*, 1992, pp. 342–354.

[14] E. Clarke, A. Gupta, and O. Strichman, "SAT-based counterexample-guided abstraction refinement," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 22, no. 7, pp. 1113–1123, Jul. 2004.

[15] P. Bjesse and J. Kukula, "Using counter example guided abstraction refinement to find complex bugs," in *Proc. Des. Autom. Test Eur.*, 2004, pp. 156–161.

[16] S. Safarpour and A. Veneris, "Abstraction and refinement techniques in automated design debugging," in *Proc. Des. Autom. Test Eur.*, 2007, pp. 1182–1187.

[17] H. Mangassarian, A. Veneris, S. Safarpour, M. Benedetti, and D. Smith, "A performance-driven QBF-based iterative logic array representation with applications to verification, debug and test," in *Proc. Int. Conf. CAD*, 2007, pp. 240–245.

[18] A. Suelflow, G. Fey, R. Bloem, and R. Drechsler, "Using unsatisfiable cores to debug multiple design errors," in *Proc. Great Lakes Symp. VLSI*, 2008, pp. 77–82.

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

[20] S. Graf and H. Saidi, "Construction of abstract state graphs with PVS," in *Proc. Comput. Aided Verification*, 1997, pp. 72–83.

[21] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking," *J. ACM*, vol. 50, no. 5, pp. 752–794, Sep. 2003.

[22] H. Jain, D. Kroening, N. Sharygina, and E. Clarke, "Word level predicate abstraction and refinement for verifying RTL Verilog," in *Proc. Des. Autom. Conf.*, 2005, pp. 445–450.

[23] P. Chauhan, E. M. Clarke, J. H. Kukula, S. Sapra, H. Veith, and D. Wang, "Automated abstraction refinement for model checking large state spaces using SAT based conflict analysis," in *Proc. Int. Conf. Formal Methods CAD*, 2002, pp. 33–51.

[24] R. Brayton, G. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*. Norwell, MA: Kluwer, 1984.

[25] A. Veneris and M. Abadir, "Design rewiring using ATPG," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 21, no. 12, pp. 1469–1479, Dec. 2002.

[26] Y. Chen and F. Chen, "Algorithms for compacting error traces," in *Proc. ASP Des. Autom. Conf.*, 2003, pp. 99–103.

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

[28] S.-J. Pan, K.-T. Cheng, J. Moondanos, and Z. Hanna, "Generation of shorter sequences for high resolution error diagnosis using sequential SAT," in *Proc. ASP Des. Autom. Conf.*, 2006, pp. 25–29.

[29] OpenCores.org, 2008. [Online]. Available: www.opencores.org

[30] M. S. Abadir, J. Ferguson, and T. Kirkland, "Logic verification via test generation," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 7, no. 1, pp. 172–177, Jan. 1988.

[31] A. Ling, S. Brown, J. Zhu, and S. Safarpour, "Towards automated ECOs in FPGAs," in *Proc. Int. Symp. Field-Programm. Gate Arrays*, 2009, pp. 3–12.

[32] Y.-S. Yang, S. Sinha, A. Veneris, and R. Brayton, "Automating logic rectification by approximate SPFDs," in *Proc. ASP Des. Autom. Conf.*, 2007, pp. 402–407.

[33] N. S. N. Een, "An extensible SAT-solver," in *Proc. Int. Conf. Theory Appl. Satisfiability Test.*, 2003, pp. 333–336.

[34] T. Cormen, C. Leierson, and R. Rivest, *Introduction to Algorithms*. Cambridge, MA: MIT Press, 1990.

Sean Safarpour (S'01) received the B.A.Sc. degree in computer engineering from the University of British Columbia, Vancouver, BC, Canada, and the M.A.Sc. and Ph.D. degrees in computer engineering from the University of Toronto, Toronto, ON, Canada.

He is currently the Chief Technology Officer with Vennsa Technologies, Inc., Toronto, where he is in charge of research and development. His research interests include design debugging, formal verification techniques, and formal engines such as SAT, QBF, and SMT solvers. He is the author of over 20 papers and 3 patents since 2004.

Andreas Veneris (S'96–M'99–SM'05) received the Diploma in computer engineering and informatics from the University of Patras, Patras, Greece, in 1991, the M.S. degree in computer science from the University of Southern California, Los Angeles, in 1992, and the Ph.D. degree in computer science from the University of Illinois, Urbana, in 1998.

In 1998, he was a Visiting Faculty with the University of Illinois until 1999 when he joined the Department of Electrical and Computer Engineering and the Department of Computer Science, University of Toronto, Toronto, ON, Canada, where he is currently an Associate Professor. His research interests include computer-aided design for debugging, verification, synthesis and test of digital circuits/systems, and combinatorics. He is the author of one book and is the holder of three patents.

Dr. Veneris has received several teaching awards and a best paper award. He is a member of the Association for Computing Machinery, the American Association for the Advancement of Science, the Technical Chamber of Greece, Professionals Engineers Ontario, and The Planetary Society.