Managing Complexity in Design Debugging with Sequential Abstraction and Refinement

Brian Keng[1], Andreas Veneris[1,2]

***Abstract*—Design debugging is becoming an increasingly difficult task in the VLSI design flow with the growing size of modern designs and their error traces. In this work, a novel abstraction and refinement technique for design debugging is presented that addresses two key components of the debugging complexity, the design size and the error trace length. The abstraction technique works by under-approximating the debugging problem by removing modules of the original design and replacing them with simulated values of the erroneous circuit. After each abstract problem is solved, the refinement strategy uses the resulting UNSAT core to direct which modules should be refined. This refinement strategy is extended by allowing refinement of across time-frames in addition to modules. Experimental results show that the proposed algorithm is able to return solutions for all instances compared to only 41% without the technique demonstrating the viability of this approach in tackling real-world debugging problems.**

I. Introduction

When functional verification [1], [2] detects a failure, it returns an *error trace* demonstrating some erroneous behavior in the implementation of the design. Design debugging is the process that follows that aims to determine the root cause of the observed failure. Today, this predominantly manual process is becoming more difficult as design blocks reach millions of synthesized gates and error traces grow to thousands of clock cycles long [3]. Due to this fact, the academic community has placed increased emphasis on automating the growing problem of design debugging which can consume up to 60% of the total verification time [4].

Recent advances in formal and SAT-based automated debugging techniques [5], [6] have achieved great strides in developing solutions that can produce scalable and higher quality results compared to earlier methods [7]. The complexity of SAT-based debugging depends primarily on three factors: design size, error trace length, and number of errors to be found (also known as *error cardinality*). The solution space grows exponentially with the error cardinality in terms of the design size and error trace length [8]. Previous work in automated design debugging aims at managing this complexity by tackling individual components independently [9]–[12]. Among the most promising of these works are abstraction and refinement techniques [9].

Abstraction and refinement [13]–[15] in the context of verification has provided a scalable and robust method for tackling large industrial instances. Previous work in debugging [9] used a similar approach to manage the design size by abstracting nodes in the design to generate a simpler abstract model. To achieve complete results, the refinement technique requires an increased error cardinality due to the fact that a single error may be mapped to multiple ones in the abstract model. This formulation guarantees that the algorithm will return a superset of the error locations that would have been found in the concrete model. However, the need for a high error cardinality for the method to work inevitably has a direct impact on its performance.

This work introduces a robust sequential abstraction and refinement algorithm for design debugging that manages both the design size and the error trace length without the need to increase the error cardinality that impacts the underlying complexity exponentially. By managing two key parameters of the debugging complexity without the need for an increase in the error cardinality, the problem size can be significantly reduced allowing larger instances to be solved.

The algorithm begins by generating an abstract problem that is a strict under-approximation of a SAT-based debugging formulation. Using simulated values of the erroneous circuit, modules of the design are abstracted reducing the complexity of the design size. After the abstracted problem is solved, the refinement strategy uses UNSAT cores to direct which modules are to be refined avoiding the need for a costly increase in the error cardinality. An extension of this refinement strategy is also presented in which UNSAT cores are utilized to direct not only which modules should be refined, but also at what clock cycles in the error trace they should be refined. This provides an effective way to manage the final component of the debugging complexity, that is, the error trace length.

Experimental results show the efficacy of this technique compared to previous work on large industrial hardware designs with long error traces from `OpenCores` [16] and from our industrial partners. The proposed abstraction and refinement technique is able to return solutions for 19 of the 22 instances compared to only 9 when the technique is not used. The extension provides further benefits as all 22 instances return solutions at the cost of increased refinement iterations.

The remaining sections of the paper are organized as follows. Section II provides notation and background information. The proposed abstraction and refinement technique is presented in Section III. Section IV present experimental results and Section V concludes this work.

II. Preliminaries

A. Notation and Definitions

For sequential circuit $\mathcal{C}$, let l denote the set of all nets in the circuit. Let x, y and s be a subset of these nets referring to the set of primary inputs, primary outputs, state elements respectively. For each $z \in \{x, y, s, l\}$, let z^i denote the corresponding Boolean vector for the i^{th} clock-cycle, or *time-frame*, for the operation of $\mathcal{C}$. Furthermore, let z^i_j denote the j^{th} indexed bit in the i^{th} time-frame for the Boolean vector z^i. The transition relation for sequential circuit $\mathcal{C}$ is denoted by $T(s^i, s^{i+1}, x^i, y^i)$. For brevity, we will denote the replicated transition relation for the i^{th} clock-cycle as T^i.

The transition relation can be written in terms of modules or components corresponding to subcircuits of $\mathcal{C}$ (e.g.

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

a Verilog module) such that the outputs of a module are only a function of its inputs. Given module q and time-frame i, let $mx_q^i, my_q^i \subseteq l^i$ denote the corresponding Boolean vector of module inputs and module outputs. A module, $M_q(mx_q^i, my_q^i)$, describes constraints such that when applying inputs mx_q^i, outputs my_q^i will be generated. Logic that is not contained within any module is denoted by ϕ_{glue}. This definition can be applied recursively for each level of the design hierarchy.

Let X^i, Y^i and S^i denote a predicate for the i^{th} clock cycle for the set of primary inputs, primary outputs and state elements respectively. An *error trace* consists of an initial state predicate, a vector of primary input predicates and a vector of *correct* or *expected* primary output predicates for the length of the error trace. It shows how the erroneous design can be driven by an initial state and primary input stimulus to generate outputs that differ from the expected behavior. The error trace can also be used to simulate C and generate internal simulated values for all nets. In particular, they can be used to generate a predicate MX_q^i and MY_q^i corresponding to a module q's input and output values for time-frame i.

B. Design Debugging

When verification fails and produces an error trace, design debugging aims to find all sets of error locations, or *suspects*, which could potentially explain an observed erroneous behavior. The set of suspects contains the actual and/or equivalent error locations for a fixed *error cardinality* [7] used by the algorithm.

SAT-based design debugging [5] encodes the problem into a SAT instance where each satisfying assignment corresponds to a set of suspects. All solutions can be found by iteratively blocking previously found solutions. The SAT instance is created in several steps. First, the transition relation is enhanced with a set of candidate error models. Each error model has a corresponding *suspect variable*, e_i, that corresponds to the i^{th} potential error location. The enhanced transition relation with the addition of the suspect variables, $e = \{e_0, \ldots, e_n\}$, is denoted by $T_{en}(s^i, s^{i+1}, x^i, y^i, e)$ (T_{en}^i for the i^{th} clock-cycle). Next, T_{en} is replicated as a time-frame expanded model for the length of the error trace. The error trace predicates are then applied to the initial state, input and output variables of the replicated enhanced transition relation. Finally, the error cardinality constraint (Φ_N) is added by constraining the sum of all suspect variables to a constant N to denote the search for N simultaneous errors. The design debugging problem can be encoded in the following SAT formula:

$$Debug_0^k = S^0 \wedge \Phi_N \wedge \bigwedge_{i=0}^{k} X^i \wedge Y^i \wedge T_{en}^i \tag{1}$$

The solution space of the SAT-based debugging formulation grows exponentially in the error cardinality as follows:

$$(combinational\ circuitry\ * \#\ trace\ cycles)^{\#\ errors} \tag{2}$$

Example 1 *Figure 1 shows the SAT-based debugging formulation unrolled for three time-frames for a simple sequential circuit with five gates and two state elements where $\otimes$ denotes the error model. An error trace constraining the initial state, inputs and outputs is used that demonstrates a failure in the third time-frame on variable y_0^2. The suspect variables for each error model are denoted by e_i corresponding to gate g_i. For $N = 1$, the gates g_0, g_2, g_3, g_4 will be returned as functionally equivalent suspects that could fix the observed failure.*

C. Abstraction and Refinement in Design Debugging

In the context of design debugging, we define an abstraction as an under-approximation if the solutions returned by the abstract problem are a subset of the concrete problem. Abstraction and refinement has recently been applied to design debugging [9] which abstracts module output nodes in the design by disconnecting them from their fan-in. This allows the unused fan-in cone to be removed simplifying the circuit. A predicate is then applied to these nodes to constrain them with the simulated values of the erroneous circuit to create the abstract problem. Suspects corresponding to abstract nodes in the problem are considered spurious and require refinement. After all solutions are found, refinement restores the abstract nodes corresponding to the spurious suspects. To achieve complete results, the error cardinality in the abstract problem must be increased by the total number of suspect variables corresponding to abstract nodes which adversely affects performance shown in Equation 2. This step is necessary because a single error in the concrete problem may map to multiple errors in the abstract one. This can occur if the propagation path of an error passes through multiple abstract nodes requiring a higher cardinality solution. The algorithm terminates when no more spurious suspects are found and guarantees to return a superset of the suspects in the concrete problem.

III. Abstraction and Refinement

A. Abstraction Formulation

The abstract instance is generated by starting off with the concrete formula from Equation 1 and adding selected constraints that will simplify it. The simplified formula offers the benefit of reducing the solve time and memory footprint compared to the concrete one. The key insight in this formulation is that the initial state and input vectors in the error trace can imply a great deal of additional information. The initial state and input stimulus, along with the erroneous circuit, provide the ability to generate the simulated value of any net in the erroneous circuit at any time-frame. We will use these values to constrain the debugging formula to generate an abstract instance.

We create an abstract debugging instance by applying, for each module q, the simulated input (MX_q^i) and output (MY_q^i) predicates to Equation 1:

$$InitAbsDebug_0^k = S^0 \wedge \Phi_N \wedge \bigwedge_{i=0}^{k} X^i \wedge Y^i \wedge T_{en}^i \wedge \bigwedge_{q \in modules} MX_q^i \wedge MY_q^i \tag{3}$$

This formulation differs from the previous work in [9] in that both the module inputs and outputs are constrained with the simulated values versus just the outputs. This subtle difference results in a strict under-approximation for the abstract instance. This is in contrast to [9] where only the outputs are constrained with their unused fan-in cone removed. As a result, there is no back propagation from these constraints which can lead to additional solutions not found in the concrete problem. The next lemma states the under-approximation property of Equation 3 formally.

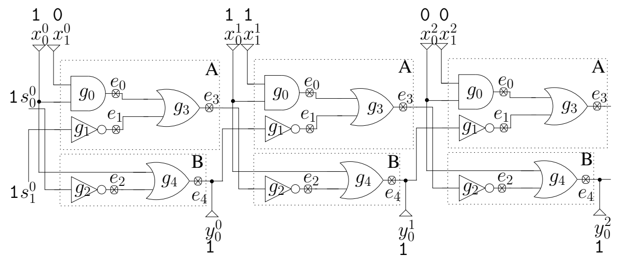

Fig. 1. Example 1: SAT-based debugging

Lemma 1 *$InitAbsDebug_0^k$ is an under-approximation of $Debug_0^k$.*

Proof: Since $InitAbsDebug_0^k$ only adds additional constraints, it will result in a subset of the satisfying assignments to $Debug_0^k$. Thus, an under-approximation by definition. ■

Although Equation 3 appears to add additional constraints, these constraints in fact allow a simplification of the equation. Applying the module input and output predicates effectively satisfies all the constraints for the corresponding modules. By removing the constraints for each module in the enhanced transition relation, Equation 3 can be re-written as:

$$AbsDebug_0^k = S^0 \wedge \Phi_N \wedge \bigwedge_{i=0}^{k} X^i \wedge Y^i \wedge \phi_{glue}(l^i, e) \wedge \bigwedge_{q \in modules} MX_q^i \wedge MY_q^i \tag{4}$$

The intuition behind this simplification is that, if the simulated input and output predicates of a module are applied, then the constraints for the module are redundant because the applied predicates define its behavior. This results in Equation 4 returning the same set of solutions as before the simplification, also stated by the following lemma:

Lemma 2 *$AbsDebug_0^k$ will return the same solutions as $InitAbsDebug_0^k$.*

Proof: $AbsDebug_0^k$ contains the same constraints as $InitAbsDebug_0^k$ except with the module constraints removed. This means that $AbsDebug_0^k$ will return a superset of solutions of $InitAbsDebug_0^k$.

Now we show that any solution found in $AbsDebug_0^k$ will be found in $InitAbsDebug_0^k$. By using the same satisfying assignment as $AbsDebug_0^k$, all the clauses except for the module constraints can be satisfied in $InitAbsDebug_0^k$. However for each module q, if there are no active suspect variables inside the module, then the module constraints, M_q^i, are fully satisfied because with the applied module input predicate, MX_q^i, the constraints amount to simulation of the module to generate the values in the module output predicate, MY_q^i. On the other hand, if there are active suspect variables inside the module, then some of the variables for the corresponding error locations will be disconnected from their fan-in constraints and become free variables. By selecting these free variables to correspond to the simulated values of these nets, the constraints will also be satisfied in a similar manner. In either case, all the module constraints will be satisfied so $InitAbsDebug_0^k$ will always be SAT when $AbsDebug_0^k$ is SAT. ■

Equation 4 gives a strict under-approximation of the debugging instance that is greatly reduced in size by the removal of module constraints. Although it has significantly less flexibility than the original formulation in Equation 1, it provides a succinct under-approximation that can be incrementally refined as described in the following subsections.

B. Module Refinement

The under-approximated abstraction from Equation 4 is significantly more constrained than the original debugging formulation in Equation 1. The fully abstracted problem will likely find very few suspects because the removed modules prevent the effect of the error models from propagating to the primary outputs. Thus only suspects that have a direct propagation path to the erroneous output will be found.

The refinement strategy selects a subset of abstracted modules and restores them by removing their module input and output predicates and adding back their original constraints. To select the subset of abstracted modules, an UNSAT core from the solved abstract instance is utilized. Intuitively, the UNSAT core represents the error propagation path backwards from the erroneous output. This path in the concrete instance will pass through many modules. In the abstract one, it will stop at one or more of the abstracted modules.

The refinement strategy uses the UNSAT core to incrementally refine modules on the error propagation path until all modules along the path have been refined. This provides a significant benefit over the refinement strategy from [9] which depends on solutions returned from the abstract problem for refinement. This is due to the fact that the use of solutions for refinement requires a formulation with higher error cardinality because an error can propagate along two separate paths. This is in contrast to using an UNSAT core which will implicitly return multiple propagation paths if they are responsible for the erroneous output.

To extract relevant information from the UNSAT core, it is traversed and clauses from any abstracted module input or output predicates are found. The modules corresponding to those clauses are marked for refinement in the next iteration. On the other hand, if there are no clauses involving module input or output predicates, then we can safely conclude that we have finished solving the problem.

The intuition behind this termination condition is that, since it does not involve any abstracted components, it will also exist in the original debugging formulation from Equation 1. Thus, no more solutions would have been found in Equation 1 either. Given this condition, a stronger result can be implied that the current refined instance contains enough information to find the same equivalent solutions as the concrete one. This is stated formally in the next theorem.

Theorem 1 *Let $RefDebug_0^k$ be the refined instance of $AbsDebug_0^k$ after zero or more refinement iterations, and U be an UNSAT core generated after blocking all satisfying assignments to solutions for $RefDebug_0^k$. If U does not contain any module input or output predicate clauses then the solutions found for $RefDebug_0^k$ will be exactly the solutions found for the entire debugging instance $Debug_0^k$.*

Proof: Since $RefDebug_0^k$ is a refined instance of $AbsDebug_0^k$, it contains fewer abstracted modules which still results in an under-approximation of $Debug_0^k$. So by definition it still returns a subset of solutions.

Now we show that any suspect found in $Debug_0^k$ will be found in $RefDebug_0^k$ given the above conditions (*i.e.,*

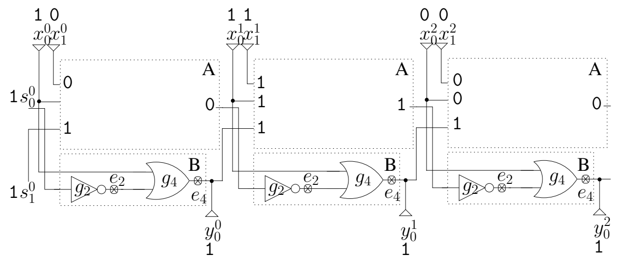

Fig. 2. Example 3: Refinement

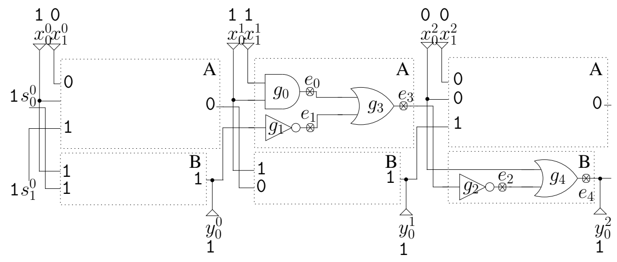

Fig. 3. Example 4: Sequential Refinement

$RefDebug_0^k$ will generate a superset of solutions of $Debug_0^k$). Assume towards a contradiction that $Debug_0^k$ has a solution, denoted by $V(e)$, that is not found in $RefDebug_0^k$ and U is the resulting UNSAT core given above. Let $blocking(e)$ denote the blocking clauses from $RefDebug_0^k$. We know $blocking(e)$ does not block $V(e)$ from being found in $Debug_0^k$ or else $RefDebug_0^k$ would have found $V(e)$. Therefore, $Debug_0^k \wedge V(e) \wedge blocking(e)$ is SAT. However, the UNSAT core U only contains clauses from blocked solutions, error trace predicates, glue logic or error cardinality constraints since it does not contain any module input or output predicate clauses. This means the UNSAT core U is a subset of the clauses in $Debug_0^k \wedge V(e) \wedge blocking(e)$ which is shown to be SAT, leading to a contradiction. So it must be the case that $RefDebug_0^k \wedge V(e)$ is SAT. ■

Theorem 1 gives us a condition to stop the abstraction refinement process without the need to refine every module. It also shows that this process will eventually provide complete results because in the worst case the abstract problem will degenerate to Equation 1 through refinement.

Example 2 *Using the proposed abstraction technique, Figure 2 the debugging instance from Example 1 has been abstracted and refined once using the proposed technique. Solving the abstract problem returns an UNSAT core involving module B output constraints and the observed erroneous primary output variable y_0^2. Following the refinement strategy, module B is refined and the resulting debugging instance is shown in Figure 2. This instance can be solved returning a subset of the exact solutions g_2 and g_4.*

C. Extension to Basic Scheme

It is possible to provide a more fine-grain refinement strategy compared to the one presented earlier while still maintaining an under-approximated abstraction. Since an UNSAT core simply contains clauses involved in unsatisfiability, the clauses can be mapped back to both their corresponding module and time-frame. Utilizing both of these pieces of information allows us to refine in a more detailed manner.

This is achieved by traversing the UNSAT core and determining which clauses correspond to added module input and output constraints. For those constraints, we determine both the corresponding module and time-frame instead of just the module. Using this strategy, modules do not need to be refined across all of time-frames but only the ones directly involved with the error. This allows us to manage the trace length by only refining modules that are necessary to debug the given problem.

This strategy allows fewer modules across time-frames to be refined but may cause more iterations because only a small amount of the problem is being refined in each iteration. To provide a trade-off between these two metrics, a radius parameter r can be introduced that refines modules that are within r time-frames of the module input or output clause involved in the UNSAT core. This balances the number of iterations against the level of the sequential refinement. Theorem 1 still applies to this sequential refinement strategy as it just refines fewer modules in each iteration compared with the strategy introduced in Section III-B.

Example 3 *Figure 3 shows the abstract problem from after two iterations of sequential refinement. Notice that only module A in time-frame 1 and module B in time-frame 2 are refined since their module input and output constraints were the only ones in the UNSAT core after solving each iteration. Solving this refined instance produces results g_0, g_2, g_3, and g_4, that is, the exact set of solutions. After solving this instance, we can get an UNSAT core with a path from primary input variables x_0^1, x_1^1 and x_0^2 to primary output variables y_0^2. According to Theorem 1, this results in a complete set of solutions.*

D. Overall Algorithm

The overall abstraction and refinement procedure is found in Algorithm 1. It begins by creating an abstract debugging instance in line 6 as described by Equation 4. Then, it iteratively solves the abstract problem and refines it in the loop of lines 7-14. After each abstract problem is solved, the UNSAT core is extracted and analyzed to determine if it involves any module input or output constraints (lines 8-10). If no constraints are found, then it terminates returning an exact set of solutions according to Theorem 1 (line 11). Otherwise, the problem is refined according to one of the refinement strategies presented in Section III-B or Section III-C. Lemma 1 is implicitly used when creating the refined problem in line 13. This step prunes previously found solutions from being considered in future iterations since we know that each abstract instance is an under-approximation.

The algorithm will always terminate because in the worst case the refinement strategy will cause the abstract problem to degenerate to Equation 1 resulting in a true condition in line 10.

IV. Experiments

This section presents the experimental results for the proposed abstraction and refinement algorithm. All experiments are run using a single core of a Core 2 quad-core 2.66 Ghz workstation with 8 GB of RAM and a timeout of 3600 seconds. Algorithm 1 is implemented for both refinement strategies and results are compared to a sequential SAT-based debugger [5] and the previous work in abstraction and refinement for design debugging [9]. PicoSAT-v913 [17] is used to solve all SAT instances and generate all UNSAT cores.

Algorithm 1 Abstraction and Refinement

```
1:  $N$ := error cardinality
2:  $e$ := set of potential suspect variables
3:  $k$ := length of error trace
4:  $sols$ := solutions found by algorithm
5:  procedure AbstractionRefinement
6:      $inst \leftarrow AbsDebug_0^{k-1}(N, e)$
7:      while true do
8:          $sols \leftarrow sols \cup \textsc{SolveAll}(inst)$
9:          $U \leftarrow \textsc{ExtractUNSATCore}(inst, sols)$
10:         if $U \cap (MX \cup MY) = \emptyset$ then
11:             return $sols$
12:         end if
13:         $inst \leftarrow \textsc{Refine}(inst, U)$
14:     end while
15: end procedure
```

The effectiveness of the proposed algorithm is shown on industrial Verilog RTL designs from `OpenCores` [16] and two commercial designs (`fxu`, `s_comm`) provided by our industrial partners. Each debugging instance is created by randomly selecting a line in the RTL and inserting a typical industrial RTL error such as a wrong state transition, incorrect operator or incorrect module instantiation. The erroneous design is then run through an industrial simulator with the accompanying testbench where a failure is detected and the error trace recorded. Suspects returned by the debuggers correspond to lines in the RTL such as assignments, `if` statements, instantiations etc. It should be emphasized that a single location in the RTL typically corresponds to multiple locations in a gate-level formulation. All abstraction techniques use modules which correspond to top-level instantiations in the Verilog design.

The results for all experiments are shown in Table I. Each design is used to create an instance by inserting an error into the design. A number is appended to each design name to indicate which error is inserted. Four sets of experiments are run on each instance corresponding to each of the different debugging methods used. The first set uses the technique based on [5] which we denote as *SAT-based debugging*. The second set is run with the previous work from [9] which we denote as *Suspect Refinement*. The final two sets correspond to the abstraction and refinement procedure presented in Algorithm 1 with the two refinement strategies from Section III-B and Section III-C which we will denote as *Module Refinement* and *Sequential Refinement* respectively. Sequential refinement is used with a radius parameter of $r = 20$.

The first five columns in Table I show the instance name, number of synthesized gates, number of initial potential suspect locations, the number of modules available for abstraction and number of clock cycles in the error trace. The next 12 columns show the run-time, peak memory and number of suspects returned by each of the four sets of experiments. A * beside the number of suspects denotes that the actual inserted error was found. A TO (MO) entry in the table is used to refer to a time-out (memory-out) condition for that experiment. For TO or MO conditions, the solutions column lists the number of partial suspects returned by that method before the condition occurs.

The benefit of the proposed abstraction and refinement technique is apparent when comparing the number of instances that return solutions. SAT-based debugging is only able to return solutions for 9, or 41%, of the 22 instances while Module Refinement and Sequential Refinement are able to return solutions for 19 and 22 of the instances respectively. The excessive size of the unrolled debugging problem from Equation 1 prevents the SAT-based debugging formulation from fitting into memory while the proposed algorithm is able to deal with these instances.

Comparing the total number of solutions, we can also see the benefit of the proposed technique compared to Suspect Refinement. In almost all of the instances, Suspect Refinement returns an excessive number of solutions due to the fact that it returns a superset of the solutions from Equation 1. This is a significant benefit of Module and Sequential Refinement techniques over Suspect Refinement because an excessive number of solutions negates the benefit of automated debugging techniques to narrow down the list of potential suspects (*i.e.,* method resolution). The aggressive abstraction from the Suspect Refinement technique combined with the need for the refinement strategy to increase the error cardinality are the primary reasons for the large increase in solutions. The refinement strategy also leads to excessive run-times where 14 of the 22 instances cause a time-out due to increased error cardinality. In these cases, the error cardinality is increased in an early iteration before many of the modules are refined, hence a lower reported peak memory for Suspect Refinement.

For the 7 instances where SAT-based debugging does not cause a memory-out or time-out, the run-time performs better in some cases such as `fxu2` and worse in others (`fxu1`). Abstraction and refinement aims to target large industrial instances, so on small instances the trade-off is less beneficial.

For memory we see similar mixed results for the same 7 instances. In the case of `fxu3`, Theorem 1 is applied generating complete results where many of the modules did not need to be refined. This results in a dramatic reduction in peak memory from 7390 MB with SAT-based debugging to 3372 MB and 4181 MB with the Module and Sequential Refinement techniques. In other cases where refinement required all the modules to be restored such as `fxu2`, the peak memory is approximately the same (7813 MB vs. 7907 MB and 7970 MB). This slight increase is primarily due to the overhead for abstraction and refinement and the enabling of tracing UNSAT cores in the SAT solver. However despite this slight overhead in memory, the majority of the instances still run using the proposed algorithm where it would not be possible otherwise.

With the two proposed refinement strategies, Theorem 1 is successfully applied in one instance for Module Refinement (`fxu3`) and four instances for Sequential Refinement (`fpu2`, `fxu3`, `mem_ctrl2`, `vga3`) where the algorithm terminates without refining all modules. Since this theorem relies on UNSAT cores, it is highly dependent on what the solver returns. Sequential Refinement typically requires more iterations, therefore it has a higher chance that it will find an UNSAT core that can be applied with Theorem 1. However, for both these refinement strategies a dramatic reduction in run-time and memory is achieved when applicable.

Figure 4 shows the number of solutions returned from the abstract problem per iteration for both the proposed refinement strategies for `ac97_ctrl1`,`fpu1`,`fxu2` and `conmax1` where *mod* stands for Module Refinement and *seq* stands for Sequential Refinement. The first iteration for all instances is the same regardless of the refinement strategy because the

TABLE I
ABSTRACTION AND REFINEMENT EXPERIMENTS

Instance Info					**SAT-based Debugging [5]**			**Suspect Refinement [9]**			**Module Refinement**			**Sequential Refinement**		
instance	# gates	# suspect	# mod	# cyc	time (s)	mem (MB)	# sols	time (s)	mem (MB)	# sols	time (s)	mem (MB)	# sols	time (s)	mem (MB)	# sols
ac97_ctrl1	30540	728	28	679	679	7560	22*	TO	1630	3076*	1135	7815	22*	TO	4003	18*
div64bits1	80907	114	2	110	234	6428	5*	302	6529	564*	209	6812	5*	889	6266	5*
div64bits2	80907	114	2	110	220	6240	5*	296	6366	564*	201	6204	5*	891	6108	5*
div64bits3	80907	114	2	110	227	6333	5*	298	6433	565*	209	6384	5*	908	6284	5*
fdct1	394133	4387	3	184	N/A	MO	0	TO	2642	2450*	N/A	MO	8*	N/A	MO	8*
fdct2	394133	4387	3	180	N/A	MO	0	TO	2520	1800	N/A	MO	8	N/A	MO	8
fpu1	83303	536	8	312	N/A	MO	0	TO	3101	879*	N/A	MO	5	N/A	MO	15*
fpu2	83303	536	8	312	N/A	MO	0	N/A	MO	2670*	N/A	MO	5*	126	1654	5*
fxu1	676928	17418	13	29	1059	7219	24*	TO	3059	1313*	720	7607	24*	777	7437	24*
fxu2	676928	17418	13	27	560	7813	18*	TO	4101	775*	790	7907	18*	830	7970	18*
fxu3	676928	17418	13	27	267	7390	1*	TO	2863	406*	183	3372	1*	196	4181	1*
mem_ctrl1	55689	2040	10	756	N/A	MO	0	N/A	MO	6153*	N/A	MO	3	TO	7648	13*
mem_ctrl2	55689	2040	10	682	N/A	MO	0	TO	780	7047*	N/A	MO	7*	114	1093	7*
rsdecoder1	16072	1100	7	114	TO	1645	101	TO	1558	178*	TO	1763	125	TO	1579	187
rsdecoder2	16072	1100	7	146	TO	2021	38	TO	1685	139	TO	1889	91*	TO	2196	99
s_comm1	1028772	27176	9	541	N/A	MO	0	TO	7916	213*	N/A	MO	17*	N/A	MO	17*
s_comm2	1028772	27176	9	541	N/A	MO	0	TO	7909	130*	N/A	MO	24*	N/A	MO	24*
vga1	175999	902	7	887	N/A	MO	0	N/A	MO	11*	N/A	MO	0	N/A	MO	14*
vga2	175999	902	7	504	N/A	MO	0	TO	7854	235*	N/A	MO	0	N/A	MO	3
vga3	175999	902	7	560	N/A	MO	0	TO	998	30497*	N/A	MO	0	610	2880	3
conmax1	140892	891	26	670	N/A	MO	0	N/A	MO	3	N/A	MO	3	TO	7284	21*
conmax2	140892	891	26	670	N/A	MO	0	N/A	MO	3	N/A	MO	3	N/A	MO	20*

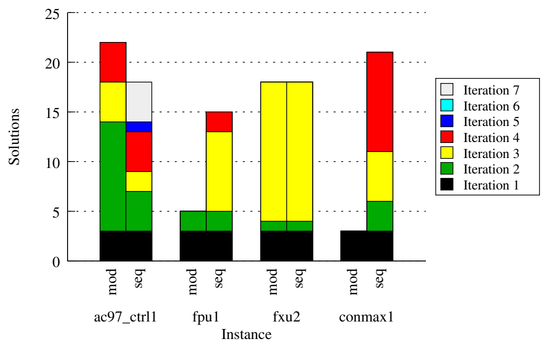

Fig. 4. Solutions per iteration for `ac97_ctrl1`,`fpu1`,`fxu2`,`conmax1`

abstract problems are identical. Benchmark `ac97_ctrl1` shows a case where the refinement of modules across all time-frames is beneficial. In the second iteration of this instance, Module Refinement finds 11 solutions versus 4 for Sequential Refinement. This demonstrates that many of the suspects require propagation across many time-frames, a disadvantage for Sequential Refinement which takes many iterations to achieve the same result. Cases `fpu1` and `conmax1` on the other hand, show great benefits from using Sequential Refinement. In these cases, the refinement of modules across all time-frames causes the instance to become too large to fit into memory causing a memory-out condition for Module Refinement. This is not an issue for Sequential Refinement which completes more iterations and finds more solutions.

V. CONCLUSION

Design debugging today is a bottleneck in the verification cycle and automation is needed to alleviate its pain. In this work, a novel abstraction and refinement algorithm is presented for design debugging to manage two key components of the problem complexity, the design size and the error trace length. This is accomplished by creating an under-approximate abstract problem that uses simulated values to replace modules of the original design. Following this, UNSAT cores are used to direct which modules to refine. An extension of the basic refinement strategy utilizes the UNSAT core to refine modules across time-frames as well. Experimental results on industrial designs show major benefits in run-time and peak-memory compared to previous state-of-the-art techniques.

REFERENCES

[1] J. Bergeron, *Writing Testbenches: Functional Verification of HDL Models, Second Edition*. Kluwer Academic Publishers, 2003.
[2] N. Amla, X. Du, A. Kuehlmann, R. Kurshan, and K. McMillan, "An analysis of SAT-based model checking techniques in an industrial environment." in *CHARME*, 2005, pp. 254–268.
[3] K.-H. Chang, V. Bertacco, and I. L. Markov, "Simulation-based bug trace minimization with BMC-based refinement," in *ICCAD*, 2005, pp. 1045–1051.
[4] H. Foster, "Assertion-based verification: Industry myths to realities (invited tutorial)," in *Computer Aided Verification*, 2008, pp. 5–10.
[5] 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.
[6] O. Sarbishei, M. Tabandeh, B. Alizadeh, and M. Fujita, "A formal approach for debugging arithmetic circuits," in *IEEE Trans. on CAD*, vol. 28, no. 5, May 2009, pp. 742–754.
[7] S. Huang and K. Cheng, *Formal Equivalence Checking and Design Debugging*. Kluwer Academic Publisher, 1998.
[8] A. Veneris and I. N. Hajj, "Design error diagnosis and correction via test vector simulation," *IEEE Trans. on CAD*, vol. 18, no. 12, pp. 1803–1816, 1999.
[9] S. Safarpour and A. Veneris, "Automated design debugging with abstraction and refinement," *IEEE Trans. on CAD*, vol. 28, no. 10, pp. 1597–1608, 2009.
[10] B. Keng and A. Veneris, "Scaling VLSI design debugging with interpolation," in *Formal Methods in CAD*, 2009.
[11] S. Safarpour, A. Veneris, and F. Najm, "Managing verification error traces with bounded model debugging," in *ASP Design Automation Conf.*, 2010.
[12] A. Suelflow, G. Fey, R. Bloem, and R. Drechsler, "Using unsatisfiable cores to debug multiple design errors," in *Great Lakes Symp. VLSI*, 2008.
[13] 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 *Formal Methods in CAD*, 2002, pp. 33–51.
[14] K. L. McMillan and N. Amla, "Automatic abstraction without counterexamples," in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer-Verlag, 2003, pp. 2–17.
[15] 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.
[16] OpenCores.org, "http://www.opencores.org," 2007.
[17] A. Biere, "PicoSAT essentials," *JSAT*, vol. 4, no. 2-4, pp. 75–97, 2008.