Efficient Debugging of Multiple Design Errors

Brian Keng[1], Duncan Exon Smith[2], Andreas Veneris[1,3]

***Abstract*—After functional verification detects a failure, design debugging aims to find all locations in the design that could be responsible for the observed error. The task of debugging becomes more difficult in modern designs because of the presence of multiple design errors. Multiple design errors exponentially increase the solution space of the debugging problem, leading to an intractable problem. This work aims to manage the complexity of multiple design errors within existing automated design debugging frameworks by using unsatisfiable cores to reduce the solution space. It builds upon previous work to generalize the generation and application of unsatisfiable cores for this purpose. An iterative debugging algorithm is presented in which unsatisfiable cores are generated as a by-product of the solving process to aid in reducing the search space for multiple design errors. Experiments on large designs for multiple errors show an average reduction in run-time of 22% with minimal impact to peak memory.**

I. Introduction

Functional verification has become a significant bottleneck in the modern VLSI design cycle [1]. An ever growing component of functional verification is when an failure occurs and the root-cause of the problem must be determined. This process, known as *design debugging*, is estimated to take up to 60% of the total functional verification time [2]. Both the time and complexity of debugging are difficult to estimate at the beginning of the process. This creates large uncertainties surrounding tight deadlines with fixed budgets. To alleviate this growing uncertainty, automation is necessary to manage and reduce this component of functional verification.

Automated design debugging techniques involve taking a counter-example from a verification failure and returning a set of locations in the buggy design that could possibly be responsible for the observed error(s). Many different techniques [3], [4] have been proposed but recently algorithms based on satisfiability (SAT) engines [5], [6] have shown to be the most promising. These algorithms translate the debugging problem into a satisfiability instance where the solutions correspond to possible locations that can be corrected in the design. Improvements in SAT-based techniques [7]–[9] have focused on three main areas that contribute to the complexity of debugging: design size, counter-example length, and number of design errors (or *error cardinality*). The error cardinality is perhaps the most daunting of the three because the solution space of the debug problem grows exponentially with the number of errors [4].

A previous algorithm [9] using multiple unsatisfiable (UNSAT) cores has shown to be effective in tackling this problem. Each core intuitively represents a tree of paths in the circuit that is responsible for causing the observed error. By analyzing the intersections of these cores, the search space can be dramatically reduced, improving the performance of the debugging engine. However, the main benefit of the technique for multiple design errors relies on the ability to find multiple overlapping UNSAT cores. This is a very difficult problem with no efficient solution in general [10]. In practice, only disjoint UNSAT cores can be found which limits the technique's ability to be more widely applicable.

In this work, we present a new algorithm to deal with debugging multiple design errors using unsatisfiable cores. It generalizes previous work by showing how to both generate and apply unsatisfiable cores for multiple design errors. The process begins by generating an UNSAT core at error cardinality zero with an unsatisfiable debug instance. The core is used to constrain the search space at the next cardinality, which is then solved for all solutions. After these solutions are blocked, another UNSAT core is generated from this instance. This process is repeated at each subsequent cardinality. By using the cores generated at each cardinality, we can rule out locations that cannot be part of solutions at higher error cardinalities. This greatly reduces the exponential search space of the debugging problem with multiple design errors. In addition, since the solving process generates these cores as a by-product, there is little overhead needed to gain significant benefit in reducing the search space.

Experimental results on large hardware designs from OpenCores [11] shows the efficacy of the proposed algorithm compared to previous work. For finding all equivalent errors up to error cardinality three, the core technique is able to reduce the total run-time on average by 22% with a negligible impact on peak memory.

The remaining sections are organized as follows. Section II describes background material. Section III describes the main contribution while Section IV presents the experimental results. Finally, Section V concludes this work.

II. Preliminaries

A. Design Debugging

Design debugging aims to find all sets of error locations, or *suspects*, which could potentially be responsible for the observed failure during verification [3]. SAT-based design debugging [5] formulates this problem as a SAT instance for a given counter-example and number of errors (or *error cardinality*). The solutions to this SAT instance correspond to all possible sets of error locations for the given error cardinality. The SAT instance is constructed in several steps. First, the conjunctive normal form (CNF)-translated design is enhanced with an error model for each location that could potentially be erroneous. This is denoted by T_{en}. Each error model has an associated *suspect variable* that when active, disconnects that location's fan-out from its fan-in and allows it to be free. Next, the combinational component of the enhanced design is copied (or unrolled) for the length of the counter-example to model the circuit behavior. On the unrolled model, the initial state (S^0), vector of inputs (X), and vector of

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

[2]Vennsa Technologies, Inc., Toronto, ON M5V 3B1 (duncan@vennsa.com)

[3]University of Toronto, CS Department, Toronto, ON M5S 3G4

expected outputs (Y) are constrained according to the counter-example. Finally, constraints (denoted by $\Phi(N)$) are added to restrict the number of active suspect variables to N. This forces the instance to search for exactly N design errors. This is formally written in the next equation for a given counter-example of length 0 to k:

$$Debug(N) = S^0 \wedge \Phi(N) \wedge \bigwedge_{i=0}^{k} X^i \wedge Y^i \wedge T_{en}^i \qquad (1)$$

Typically, to find multiple design errors, Equation 1 is iteratively solved from $N = 1$ to a maximum desired error cardinality. By blocking all previously found lower cardinality solutions, one can find all minimal cardinality equivalent design errors [4]. In this work, we use this methodology as a basis for finding multiple design errors.

B. Unsatisfiable Cores in Design Debugging

An unsatisfiable (UNSAT) core is a unsatisfiable subset of clauses of an unsatisfiable Boolean formula written in CNF. Modern SAT-solvers can produce UNSAT cores as a by-product of their solving process [12]. In the context of design debugging, an UNSAT core intuitively represents a tree of paths in the circuit from which the primary inputs and initial states propagate to the expected primary outputs and result in a conflict.

This idea has been used in several algorithms to generate information to aid in the process of design debugging [7]–[9]. The work in [9] uses the above concept of UNSAT cores to restrict which areas of the circuit need to be examined. If an UNSAT core exists, at least one of the suspect variables in the core must be activated to break the conflict. Given multiple UNSAT cores, this idea can be extended to further restrict the solution space.

Deriving such an UNSAT core can be done by setting all the suspect variables in Equation 1 to `0`. Since the simulated outputs of the circuit do not match the expected outputs (or else there is no failure), the instance is unsatisfiable and an UNSAT core can be generated. However in general, finding multiple UNSAT cores is difficult. The work in [9] finds multiple disjoint UNSAT cores by removing previously found UNSAT cores and calling the SAT-solver again. However, the main benefit for multiple design errors is if multiple overlapping UNSAT cores can be found. There are several algorithms to accomplish this goal, but they require significant computation [10]. In practice, a single UNSAT core can be generated easily while multiple disjoint cores can be found if the problem happens to contain them.

The following example demonstrates the use of a single UNSAT core to restrict the solution space for finding a single design error.

Example 1 *Figure 1 shows a visualization of the debug instance from Equation 1. A simple combinational circuit has been augmented with the error model denoted by $\otimes$ which disconnects a gate's fan-out from its fan-in to become free. Next, constraints are added for the input and expected outputs. Notice that the outputs mismatch on g_4 and g_5.*

By disabling all suspect variables, we can generate an UNSAT core involving variables e_1, e_2 and e_4 for this instance. This core implies if there is a single design error, it must be among the corresponding gates. When solving the SAT-instance for a single design error ($N = 1$), $e_2 = 1$ is the only satisfiable solution corresponding to gate g_2, confirming the information found in the UNSAT core.

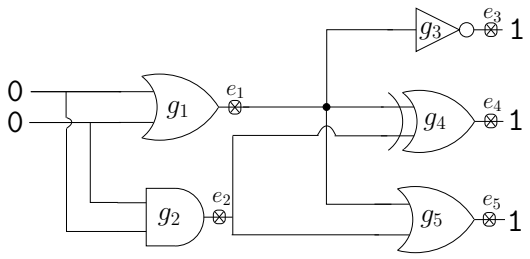

Fig. 1. SAT-based Design Debugging Instance

III. Efficient Debug of Multiple Design Errors

Unsatisfiable cores can provide valuable information in determining which locations are involved in the observed failure. However as mentioned previously, finding multiple cores is not a simple task. In this section, we describe a method for finding a single unsatisfiable core at each error cardinality that can be used to reduce the search space for higher cardinalities. The unsatisfiable cores generated are a natural result of the solving process at each cardinality and fit within existing SAT-based frameworks.

A. Reducing the Search Space of Multiple Design Errors

As previously described, an unsatisfiable core intuitively contains information regarding which parts of the design are responsible for the observed failure. Given a single core that is derived by setting all the suspect variable to `0`, we know that at least one of the suspect variables in the core must be activated in order to generate a satisfying assignment. However, this result can be generalized to higher cardinalities as well.

When debugging at a certain cardinality N, one finds all possible satisfying solutions that have exactly N suspect variables active by adding blocking clauses. Once the solver proves unsatisfiability at that cardinality, it also generates an unsatisfiable core. However intuitively, this core differs from the one described in Section II-B. Instead of only finding a single conflict, the suspect variables allow additional flexibility in breaking up to N conflicts. However, even by breaking N conflicts, there still must exist at least one more conflict that cannot be broken. To achieve a SAT result, we must be able to break each one of these conflicts. This means that some subset of suspect variables ($> N$) involved in these conflicts are needed in order to break the resulting core. This leads to the result that if a suspect variable is not included in the unsatisfiable core at N, then it cannot be in a solution at $N+1$. This is stated more precisely in the next theorem.

Theorem 1 *Let U be the unsatisfiable core generated by solving and blocking all solutions for the debug instance $Debug(N)$. If $e_i \notin U$, then $e_i = 1$ is not in any satisfying solution of the instance $Debug(N + 1)$ with all lower cardinality solutions blocked.*

Proof: Assume towards a contradiction that a suspect variable $e_i \notin U$ and $e_i = 1$ is part of a satisfying solution for $Debug(N + 1)$. Since $e_i = 1$, then exactly N other

suspect variables may be active. U is an unsatisfiable core that cannot be broken with N or less suspect variables being active. Therefore, $Debug(N+1)$ is still unsatisfiable leading to a contradiction. So it must be the case that $e_i = 1$ is not part of any satisfying solution for $Debug(N+1)$. ■

Although Theorem 1 gives us valuable information about the next cardinality, it can also provide information about higher cardinalities. The core can help us constrain the solution space of higher cardinalities by specifying a superset of the suspect variables which must be active. This is described in the next corollary.

Corollary 1 *Given U from Theorem 1, any solution found at a cardinality greater than N must have at least $N+1$ suspect variables active from U.*

Proof: If fewer than $N+1$ suspect variables from U are active, then U still forms an UNSAT core so it cannot be part of any solution for a higher cardinality. Therefore, any satisfying solution must involve at least $N+1$ suspect variables from U. ■

Notice that the results in [9] are a special case of Theorem 1 and Corollary 1 where $N = 0$. Similar results for multiple UNSAT cores can be applied for the higher cardinality cores as well. However, as mentioned before, these multiple cores are difficult to generate efficiently. The next example shows how Theorem 1 helps reduce the search space at higher cardinalities.

Example 2 *Continuing from Example 1, if we use the UNSAT core resulting from the debugging instance with $N = 1$, we will get an UNSAT core involving variables e_1, e_2, e_4 and e_5. When solving the $N = 2$ instance (while blocking previously found solutions), we will get a single solution of $e_4 = 1 \wedge e_5 = 1$ corresponding to gates g_4 and g_5. This confirms the result from Theorem 1 where the suspects in the solution must be in the UNSAT core from the previous cardinality.*

Since unsatisfiable cores are generated at each cardinality as a by-product of solving process, Theorem 1 can be used effectively with very little overhead. An integrated algorithm is described in the next sub-section.

B. Overall Algorithm

Algorithm 1 presents pseudo-code for the overall algorithm to debug multiple design errors. The algorithm takes as input the maximum error cardinality (N_{max}) and outputs solutions corresponding to possible locations for the design error(s). Lines 5-10 consist of the main loop where the debugging instance is solved for each error cardinality and an UNSAT core is generated. First, the debugging instance is created at the current error cardinality on line 6. The SAT instance is made such that any satisfying solution will activate exactly n suspect variables corresponding to the current error cardinality. Next, Theorem 1 is applied where suspects not inside the previously found core are removed from the instance (line 7). This can either by done by re-generating the instance or simply setting the corresponding suspect variables to `0`. Using this optimized instance all solutions are found (line 8). Finally, the UNSAT core is extracted on line 9.

On line 9, it is important to use the unoptimized instance (with all the solutions blocked) to derived the UNSAT core. This is due to the instance generated on line 7 removing certain suspect variables from consideration. These removed variables apply only to the current cardinality, not higher ones. If the optimized instance were used instead, it would generate a core whose suspect variables incorrectly constrain the solution space.

A more general method could use Corollary 1 instead. In this way, additional constraints could be used to force suspect variables in the core to be at least a certain number. However, too many additional clauses would be needed to model this constraint, negating the benefit of the optimization in the first place.

Algorithm 1 Debugging Multiple Design Errors

```
1:  N_max := error cardinality
2:  sols := solutions found by algorithm
3:  procedure DEBUGMULTIPLEERRORS(N_max)
4:      U ← {}, sols ← {}
5:      for n : 0 .. N_max do
6:          inst_orig ← Debug(n)
7:          inst_opt ← DISMISSSUSPECTS(inst, U)
8:          sols ← sols ∪ SOLVEALL(inst_opt)
9:          U ← EXTRACTCORE(inst_orig ∧ block(sols))
10:     end for
11:     return sols
12: end procedure
```

IV. EXPERIMENTS

In this section we present results for the proposed algorithm for debugging multiple design errors. All experiments are performed using a single core of an Intel Core i5 3.1 GHz machine with memory limit of 8GB and a timeout of 7200 seconds. The debugger used is a C++ sequential SAT-based engine based on [5]. MINISAT [13] is used to solve all the SAT instances. The UNSAT core extraction feature in the solver is turned on only during the EXTRACTCORE step in Algorithm 1. All other runs used the default solver settings with the core extraction feature off.

The effectiveness of the proposed technique is shown on a variety of RTL designs from OpenCores [11]. Each debug instance is generated by randomly selecting a line in the RTL and inserting a typical RTL design error such as a incorrect operator, state transition, or module instantiation. These RTL errors translate to multiple design errors at the gate level. To effectively demonstrate the effect of multiple design errors, error models (*i.e., suspect variables*) are placed on the output of gates of the synthesized design that correspond to signals at the RTL level. This translates to a suspect variable on each bit of all Verilog `inputs`, `wires` and `regs`. This ensures that the error models can correspond precisely to any given design error. Next, each instance is run through its accompanying testbench, simulated and the resulting counter-example is recorded. This counter-example contains the initial state and the primary input/output values for each cycle of the simulation trace, which are used to constrain the debugging problem. The instances are labeled with the circuit name followed by a number indicating different errors that were inserted.

The experimental results for Algorithm 1 with $N_{max} = 3$ are shown in Table I. The proposed algorithm is denoted by

TABLE I
DEBUGGING MULTIPLE DESIGN ERRORS EXPERIMENTS

instance info								orig [5]		core					
instance	gates	flops	clks	# sus	sols (N=1)	sols (N=2)	sols (N=3)	time (s)	mem (MB)	core (s)	total (s)	mem (MB)	dismiss (N=1)	dismiss (N=2)	dismiss (N=3)
ac97_ctrl1	15109	2482	300	2483	25	3	262	294	2466	16	165	2628	2450	2471	2413
ac97_ctrl2	15114	2483	200	2456	12	132	205	402	1937	37	303	1903	2430	2350	2333
divider1	3773	424	39	498	18	503	5228	653	487	26	693	473	136	31	9
mem_ctrl1	46767	1239	40	2858	6	14	4	57	1309	9	35	1385	2824	2705	2508
mrisc_core1	15407	2161	41	2566	52	118	616	317	913	10	245	886	2446	2378	2112
mrisc_core2	14456	1371	40	2044	61	358	3392	637	708	7	488	690	1799	1798	1514
rsdecoder1	13023	526	40	5159	21	102	2714	2518	1000	81	2124	1026	2870	1139	602
usb_funct1	10217	736	37	1593	27	223	299	1298	788	203	1451	806	1420	994	480
vga1	72292	17110	50	2936	38	328	3578	3958	1639	182	2625	1440	2766	2568	2369
vga2	73546	17213	100	1103	118	116	15	57	1119	4	37	1133	939	1065	1051

core and compared against the SAT-based debugger described in [5], denoted by *orig*. Each row of the table corresponds to a different instance that is run. The first five columns of the table show the instance name, number of combinational gates, number of state elements, length of the counter-example, and number of potential suspect locations. The next three columns show the number of solutions for each of the three error cardinalities which are identical for both *core* and *orig*.

The next two columns show the run-time in seconds and the peak memory in MB for *orig*. Columns 11-16 show the results for *core*. These six columns show the cumulative time needed to generate all the UNSAT cores, total run-time including core generation, peak memory and number of suspects removed from consideration for each of the three error cardinalities.

The benefit of the proposed technique is clearly shown when looking at the total run-time compared with the previous work. There is an average reduction 22% in the total run-time when using UNSAT cores. This comes with almost no impact to peak memory resulting in a slight reduction of 0.4% in favor of the *core* technique. This improvement in run-time can be primarily explained by the drastic reduction in the search space for each cardinality. From columns 14-16, the average reduction in terms of number of suspects across all instances are 83%, 74% and 66% for $N = 1$ to $N = 3$ respectively.

Figure 2 shows the number of dismissed suspects for several instances for each N. When the number of dismissed suspects is large, there is significant benefit to total run-time, as in the case of `vga` with a run-time reduction of 34%. However, when the number of dismissed suspects is small, as in the case of `divider1`, the benefit of reducing the solve time may not outweigh the increased overhead of finding an UNSAT core. A similar case occurs in `usb_funct1` where the trade-off of finding a core at $N = 2$ did not pay off during the solving of $N = 3$. Although these two instances show negative results, overall the technique still performs well in reducing run-time.

In terms of peak memory, the two sets of experiments returned mixed results. In general, extracting an UNSAT core causes an increase in memory usage due to the necessary bookkeeping needed to keep track of the core during solving. However, for these experiments the peak memory occurred primarily while solving the $N = 3$ case where core extraction is turned off. As a result the memory results are almost identical to each other.

V. CONCLUSION

In this work, we present a new algorithm for debugging multiple design errors efficiently. It builds upon previous work by generalizing the concept of how to generate as well as apply unsatisfiable cores to reduce the solution space of the debugging problem. The generation and application of these cores fits naturally within existing SAT-based debugging frameworks to allow easy extensibility of existing implementations. Experimental results show large improvements to run-time while having minimal impact to peak memory.

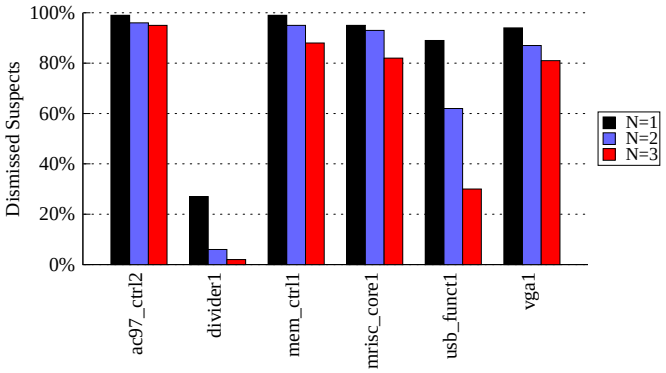

Fig. 2. Percentage of Dismissed Suspects

REFERENCES

[1] J. Bergeron, *Writing Testbenches: Functional Verification of HDL Models, Second Edition*. Kluwer Academic Publishers, 2003.

[2] H. Foster, "Assertion-based verification: Industry myths to realities (invited tutorial)," in *Computer Aided Verification*, 2008, pp. 5–10.

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

[4] 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.

[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] Y. Chen, S. Safarpour, J. Marques-Silva, and A. Veneris, "Automated Design Debugging with Maximum Satisfiability," *IEEE Trans. on CAD*, vol. 29, pp. 1804–1817, November 2010.

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

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

[9] A. Suelflow, G. Fey, R. Bloem, and R. Drechsler, "Using Unsatisfiable Cores to Debug Multiple Design Errors," in *Great Lakes Symp. VLSI*, 2008.

[10] Y. Oh, M. N. Mneimneh, Z. S. Andraus, K. A. Sakallah, and I. L. Markov, "AMUSE: a minimally-unsatisfiable subformula extractor," in *Design Automation Conf.*, 2004, pp. 518–523.

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

[12] L. Zhang, "Searching for truth: Techniques for satisfiability of Boolean formulas," Ph.D. dissertation, Princeton, 2003.

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