Verification of Large Synthesized Designs

Daniel Brand

IBM Research Division
Thomas J. Watson Research Center
Yorktown Heights, New York, U.S.A.

Abstract

The problem of checking equality of boolean functions can be solved successfully using existing techniques for only a limited range of examples. We extend the range by using a test generator and the divide and conquer paradigm.

1. Introduction

Since the synthesis process, whether automatic or manual, cannot be guaranteed to be error free, there is a need for checking its correctness. That is, we have to prove the equality F = G, where F is the input into synthesis and G is the result of synthesis. The traditional approach to this problem is to build a canonical BDD representation [6] for each function and then simply compare the BDDs for identity. Many variations and improvements have been made [2,7,10,14] so as to extend the range of applicability. The applicability of BDDs is limited because their memory requirements may grow exponentially with the size of the function. Therefore "running out of memory" is the usual failure mode of BDDs. This is more serious than the failure mode "running out of time" because it is much easier to allocate more time to a problem, than to allocate more memory. This has been recognized by [9] where correctness can be guaranteed with a reliability increasing with increasing CPU time.

One method of boolean reasoning that does not explode in terms of memory is a test generator. The most straightforward approach towards proving F = G using a test generator would form the exclusive OR, $F \oplus G$ (Figure 1), and then ask whether the output of the XOR gate is testable. If it is testable for stuck at 0 then the resulting test pattern constitutes a counter example to the assertion of functional equivalence. If it is not testable for stuck at 0 then the two functions are identical. If it is not testable for stuck at 1 then the two functions are complements of each other.

Throughout the paper we will use the term "miter" to refer to the configuration of Figure 1. In general, a miter can appear in the middle of larger logic, where it is defined to consist of a two-input XOR gate, plus the symmetric set difference between the transitive fanins of the two inputs into the XOR gate. In other words, a miter starts at an XOR gate and extends towards primary inputs till nets shared by both cones of logic.

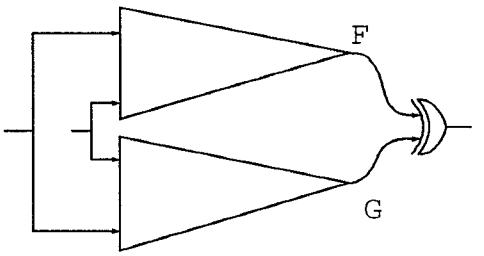

Figure 1. A miter.

The above straightforward approach is not likely to be successful because test generators tend to have a lot of difficulty with miters. While there are test generation strategies specifically designed for miters [8], they are bound to fail if the miter is big enough. The main observation of this paper is that the difficulty for a test generator is dependent mainly on the size of the miter, rather than on the size of the surrounding logic. The key to our approach is to keep applying a test generator to configurations where the size of the miters is small and independent of the size of the functions subject to verification.

One way of making the verification problem easier is to do it in stages [5]. Suppose that f is a subfunction of F(f) and g is a subfunction of G(g). We would like to first prove f = g and then use it to simplify the proof of F(f) = G(g). To do that we must first solve two problems:

1. It maybe difficult to find such subfunctions f and g because synthesis frequently does not preserve the function of internal nodes. If f was optimized subject to the don't cares of the whole F then there may be no g = f.
2. Suppose that we do find f = g; how do we take advantage of it? We cannot simply replace f and g with a new variable y. Even if f = g and F(f) = G(g) it is possible that $F(y) \neq G(y)$ because F might have been optimized subject to the don't cares of f. This problem is

referred to as "false negative", that is, f = g and F(y) ≠ G(y) need not necessarily imply that F(f) ≠ G(g).

Our solution to the first problem is not to ask whether f = g, i.e., " is f ⊕ g testable for stuck at 0?". Instead we insert an XOR gate between f and all its immediate fanouts and make g the other input into the XOR gate (see Figure 2). Then we check whether the output of the XOR gate is testable (in the context of the surrounding logic of F). Suppose that it is not testable for stuck at 0. That means that for every input pattern either f = g, or f ≠ g, but the difference cannot be propagated to primary outputs. In other words, g can replace f inside F without changing the function of F. Similarly, if the XOR gate is not testable for stuck at 1 then $\bar{g}$ can replace f.

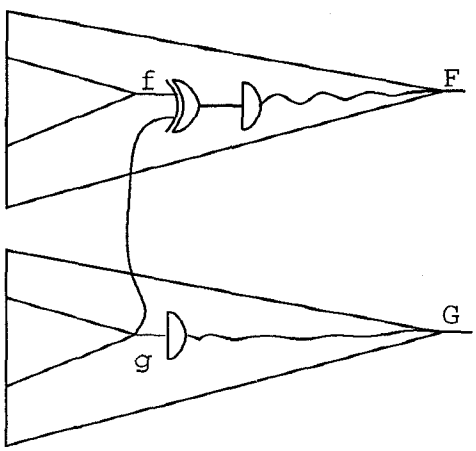

Figure 2. Can g replace f inside F?

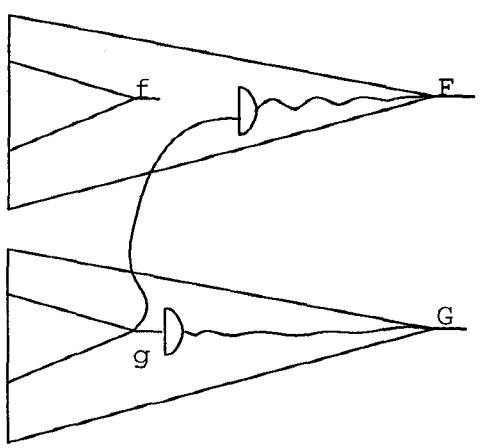

Figure 3. g replaces f.

Our solution to the second problem is to actually perform the replacement. However, it is important that the logic of g not be copied inside F, but rather F and G must share the same copy of g (see Figure 3). Proving F(g) = G(g) is easier than proving F(f) = G(g) because the miter F(g) ⊕ G(g) is smaller than F(f) ⊕ G(g).

The whole process proceeds from inputs of F to its outputs. At each step we have a subfunction f and try to find a subfunction g of G that could replace f. If we find a suitable g we make the replacement, while retaining the function of F. As we proceed F keeps resembling G more and more and the process stops at the outputs of F, which should be replaceable by the outputs of G. If an output of G cannot replace the corresponding output of F then the test generator gives an input pattern constituting a counter example.

2. Algorithm

As described in the introduction, the approach is based on the following

Lemma:

Let x be a vector of variables and y be a single variable. Consider arbitrary function f(x), g(x), F(x, y). (Since all three functions depend on x we will drop x from our notation; for example by F(f) we will mean F(x, f(x)).) Then the following two identities hold.

$$F(f) = F(g) \text{ iff } F(f \oplus g) = F(0) \quad (1)$$

$$F(f) = F(\bar{g}) \text{ iff } F(f \oplus g) = F(1) \quad (2)$$

Proof:

We will be done if we show that (1) and (2) are true for any input pattern x. For a given input x there are four possible combinations of values for f and g. For illustration we will show the case f = 0 and g = 1; all the other cases are similar. Substituting f = 0, g = 1
in (1) we get F(0) = F(1) iff F(1) = F(0)
in (2) we get F(0) = F(0) iff F(1) = F(1)
□

The right hand side of (1) says that the output of the XOR is not testable for stuck at 0 and the left hand side says that in this case we may replace f by g. Similarly (2) says that if the XOR is not testable for stuck at 1 then we can replace f by $\bar{g}$.

We now give the algorithm for proving F = G, where both functions are given by a multi-output combinational network. The algorithm is followed by a detailed explanation.

0. for each net calculate its primary inputs
1. form a list fs of nets in F in topological order
2. for each f in fs
3. form a list gs of some nets in G
4. for each g in gs
5. if F(f⊕g) = F(0) then replace f by g
6. if F(f⊕g) = F(1) then replace f by $\bar{g}$

For each of the above statements we will give an explanation as well as its computational complexity. Let the number of connections in both F and G be n. We will assume that the number of nets is also of the same order as n.

Statement 0: calculates information used for a heuristic selection of the candidates g in statement 3. It collects information whose size can be $O(n^2)$ in the worst case. Therefore its time and space complexity is $O(n^2)$.

Statement 1: The only nets required in fs are primary outputs; all other nets are placed there only for speed. The more internal nets are included in fs, the more frequently we need to invoke the test generator, but the easier will be the test generator's work because it will be working on smaller miters. Our heuristic is to have the nets in fs separated by approximately two stages of logic. In our experiments the number of stages of separation did not significantly affect the performance of the algorithm, provided for any miter that is too large we consider smaller miters in its place. The topological ordering is done in linear time.

Statement 2: There can be at most n of the nets f.

Statement 3: In contrast to fs, the choice of gs has a tremendous affect on performance. If the list gs were too long then we would call the test generator too many times even for nets f that cannot be replaced by any g. On the other hand if gs were too short then we might fail to find a g to replace a particular f, which would make the test generator's job harder later on. Our heuristic forms a list gs giving preference to nets g with a name related to the name of f, followed by nets g that have the same simulation result as f on a small number of random patterns, followed by all remaining nets.

The list gs is pruned in order to minimize the number of calls to the test generator.
- We require that each g depend on no more primary inputs than f does.
- We use approximate fault simulation to discard candidates g which are not likely to replace f (similarly as in [1]).

Then the list gs is truncated after k elements; in our experiments we used k = 20.

This statement has a worst case complexity O(n) because we may be forced to consider all n nets g. Since this statement is executed n times it contributes $O(n^2)$ to the overall algorithm.

Statement 5 and 6:: We call the test generator [11,12] to determine whether g can replace f. If so, we make the replacement and terminate the loop over gs. This statement contributes only O(n) complexity to the algorithm because we allow the test generator to run only a fixed amount of time independent of the size of the network. For each f the test generator is called at most k times.

Thus the overall worst case complexity is $O(n^2)$, but we cannot guarantee to verify every design. In order to obtain such a guarantee we would have to allow an unbounded amount of test generation time, in which case the worst case complexity would depend exponentially on the differences between the two designs.

The fact that in our experiments we did not experience an exponential blowup suggests that in synthesized logic it is normally possible to keep the miter size constant and small. This does not imply optimization based on local information only; the test generator does take into account global information far away from the miter under consideration.

3. Experimental results

Table 1 shows results on the ISCAS benchmarks [3,4]. Each benchmark was synthesized using the standard script of BooleDozer[13], which includes data flow optimization, redundancy removal, factoring, technology mapping and others. We did not run any timing optimization because it is very dependent on timing assertions and we do not anticipate that it would cause problems to our algorithm. On the other hand, to the standard script we added a transformation that collapses as large pieces of logic as possible into a two-level representation, minimizes and factors. We added this transformation because it destroys the original structure of the design and therefore makes it difficult to find an internal net g in the implementation that can replace a given net f in the specification.

For each design we give the number of connections of the two designs being compared – before synthesis and after synthesis. Then we give the CPU time in seconds on IBM RS/6000 Model 550. In the last column we give a ratio between the verification time and time to read in the two designs, because this is a measure independent of the hardware and other system variables that are not pertinent to our algorithm. (A linear algorithm would have constant relative time.)

4. Conclusions

We have shown how a test generator can be used for verification, and demonstrated its effectiveness by verifying all the ISCAS benchmarks, which to our knowledge has not been accomplished by any other approach yet.

The algorithm proved successful because there are apparently many nets in a specification for which there exists a corresponding net in the implementation. This appears true even after extensive global optimizations. It is not very surprising given the experience of transduction [15], which relies on a related phenomenon. Namely, it is very common that in a logic network there are nets computing related functions possibly for unrelated purposes.

It is possible, however, that our algorithm may fail on some designs in the future. Even then it would not be necessary for a designer to partition his design (which is a common industry practice) because our algorithm is not as sensitive to design size as it is to differences between

the two designs being compared. Therefore any partitioning, if necessary, should be in the synthesis process. That is, we can write several checkpoint files during synthesis and do verification between successive pairs of checkpoint files.

TABLE 1 -- Verification of ISCAS benchmarks

CIRCUIT	CONNECTS		CPU	RELATIVE
	BEFORE	AFTER	SEC	TIME
C17	31	19	1	20
C432	712	346	4	4
C499	1105	539	38	24
C880	1125	609	5	3
C1355	1937	903	9	3
C1908	2244	647	22	9
C2670	2783	1211	58	15
C3540	3666	1652	39	8
C5315	5346	2880	29	4
C6288	9120	4690	193	13
C7552	8107	3288	136	12
S27	26	25	1	12
S208	187	115	1	3
S298	267	175	1	2
S344	304	211	1	2
S349	308	211	1	2
S382	336	217	1	2
S386	367	193	1	2
S400	350	214	1	2
S420	373	231	5	7
S444	382	217	2	3
S510	456	409	3	3
S526	1475	309	2	2
S526N	475	288	2	2
S641	617	310	3	3
S713	668	309	3	3
S820	799	485	5	4
S832	811	485	11	8
S838	739	419	47	35
S1196	1055	849	8	3
S1238	1087	853	10	4
S1423	1260	815	8	3
S1488	1420	947	11	5
S1494	1426	939	10	4
S5378	4475	2096	30	4
S9234	8240	2903	77	6
S15850	14343	5616	192	8
S35932	30352	17954	519	8
S38417	33798	15187	614	9
S38584	34498	19669	736	9

Acknowledgements

I am grateful to Sandip Kundu and Prakash Narain for being able to use their test generator. Andreas Kuehlmann, Vijay Iyengar, Leon Stok and Louise Trevillyan read the manuscript and provided valuable suggestions.

References

[1] M.Abramovici, P.R.Menon, D.J.Miller, "Critical Path Tracing: An Alternative to Fault Simulation", IEEE Design and Test, Vol.1, February 1984, pp. 83-93.

[2] P.Ashar, A. Ghosh, S. Devadas, A.R. Newton, "Combinational and Sequential Logic Verification using General Binary Decision Diagrams", International Workshop on Logic Synthesis 1991.

[3] F. Brglez, P.Pownall, R. Humm, "Accelerated ATPG and Fault Grading via Testability Analysis", *IEEE International Symposium on Systems and Circuits*, June 1985, pp. 695-698.

[4] F. Brglez, D. Bruan, K. Kozminski, "Combinational Profiles of Sequential Benchmarks Circuits", *IEEE ISCAS Proceedings*, May 1989, pp. 1929-1934.

[5] C.L. Berman, L.H. Trevillyan, "Functional Comparison of Logic Designs for VLSI Chips", Proceedings of ICCAD 1989, pp. 456-459.

[6] R.E. Bryant, "Graph Based Algorithms for Boolean Function Manipulation", IEEE Transactions on Computers, Vol. 35, Aug. 1986, pp. 677-691.

[7] M. Fujita, H. Fujisawa, N. Kawato, "Evaluation and Improvements of Boolean Comparison Method Based on Binary Decision Diagrams", Proceedings of ICCAD 1988, pp. 2-5.

[8] P.Goel, "An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits", IEEE Transactions on Computers, Vol. C-30, March 1981, pp. 215-222.

[9] J. Jain, J. Bitner, D.S. Fussell, J.A. Abraham, "Probabilistic Design Verification", Proceedings of ICCAD 1991, pp.468-471.

[10] S-W. Jeong, B. Plessier, G. Hachtel, F. Somezi, "Extended BDD's: Trading off Canonicity for Structure in Verification Algorithms", Proceedings of ICCAD 1991, pp.464-467.

[11] R.P.Kunda, P.Narain, J.A.Abraham, B.D.Rathi, "Speed up of Test Generation using High Level Primitives", 27th Design Automation Conference, June 1990, pp. 594-599.

[12] S.Kundu, L.H.Huisman, I.Nair, V.S.Iyengar, L.N.Reddy, "A small Test Generator for Large Designs", Proceedings of International Test Conference, Sept. 1992, pp. 30-40.

[13] D.S.Kung, R.F.Damiano, T.A.Nix, D.J.Geiger, "BDDMAP: a technology mapper based on a new covering algorithm", Proceedings of DAC 1992, pp. 484-487.

[14] S. Malik, A. Wang, R. Brayton, A. Sangivanni-Vincentelli, "Logic Verification Using Binary Decision Diagrams in a Logic Synthesis Environment", Proceedings of ICCAD 1988, pp. 6-9.

[15] S. Muroga, Y. Kambayashi, H.C. Lai, J.N. Culliney, "The Transduction Method – Design of Logic Networks Based on Permissible Functions", *IEEE Transactions on Computers*, Vol 38, No. 10, October 1989, pp. 1404-1424.