Algorithms for Compacting Error Traces

Yirng-An Chen
yachen@novas.com
Novas Software Inc.
San Jose, CA 95110, U. S. A

Fang-Sung Chen
fschen@sis.com.tw
Silicon Integrated Systems Corp.
Hsin Chu, Taiwan, R.O.C.

Abstract— In this paper, we present a concept of compacting the error traces generated by pseudo-random/random simulations. The new shorter error trace not only decreases the time of user's debugging process but also reduces the simulation time required to verify the bug fixes. Two algorithms *CET1* and *CET2* are presented to perform the task of compacting the error trace. Both algorithms first use an efficient approach to eliminate the redundant states to generate the unique states of the error trace. Then, *CET1* build the connected graph of these unique states by computing the reachable states by one cycle for each unique state, and then apply *Dijkstra's* shortest path algorithm to find out the shortest error trace in the connected graph. Compared with *CET1*, *CET2* computes the reachable states by one cycle for those unique states, when they are needed in *Dijkstra's* shortest path algorithm to find the shortest error trace. After finding the shorter trace, the corresponding input/output test vectors are generated. The experimental results show that both algorithms can reduce the length of error traces dramatically for most cases using reasonable memory. For cases required longer CPU time to find the shortest trace, *CET2* is up to 37 times faster than *CET1*.

I. Introduction

The increasing complexity and circuit size of designs have made function verification task as one of bottlenecks in VLSI design cycle. In recent conferences, many panelists claimed that functional verification takes about 60%-70% of design cycle. The tasks of functional verification includes detecting errors in designs and finding the causes of the errors (debugging process). To detect errors in designs, verification engineers and designers may generate test vectors manually, write test benches using tools, and write properties. Currently, formal verification tools checks whether the design satisfies their properties. If not, a set of counter-examples ("error traces") can be generated to debug the design. Usually, formal verification tools can use techniques [4, 9] to generate good error traces to make the debugging process easier. Test vectors and test benches are mainly used in simulators and emulators to detect errors in designs. If errors are found, error traces are written out for debugging process. In general, these error traces produced by random/pseudo-random test benches contain very long cycles and many redundant states. It is very difficult for users to debug using these traces and to take long simulation time to verify the bug fixes. Thus, we are inspired to develop the technique to compact error trace generated by random/pseudo-random test benches.

The main reason for compacting error traces generated by pseudo random simulations is that they may contains very long cycles and many redundant states. In the debugging process, users need to find the causes of bugs from the information in the error trace. Thus, The longer cycles of simulation trace are, the more time users spend to debug. Moreover, after finding the causes of bugs and fixing them, users perform the same simulation run to verify the bug fixes. Then, users put the simulation run into the daily regression suites to prevent the same bugs happen again. Therefore, shorter error traces not only reduce the simulation time required to verify bug fixes, but also reduce the simulation time for daily regression.

A digital circuit can be formulated as a finite state machine (FSM) and the circuit behavior can be viewed as the sequences of state traversal. The error trace contains all information of a sequence of state traversal from an initial state to an error state. The problem of compacting the error trace can be defined as follows: Given a digital circuit design and its simulation error trace information containing the initial state and the error state, the problem is to find other error trace as shorter as possible.

Our approach to solve compacting the error trace is based on the following three techniques: First, an efficient technique is used to eliminate the redundant states and to generate the unique states of the error trace. Then, the connected graph of these unique states is generated by computing the reachable states by one cycle for each unique state. The *Dijkstra's* shortest path algorithm is applied on the connected graph to find out the shortest error trace. Finally, the corresponding input/output test patterns are automatically generated for the new error trace. This algorithm is named as *CET1*. In experiments, we found out that not all of the unique states are needed to compute their reachable next states. Thus, algorithm *CET2*, modified from *CET1*, only computes the reachable states by one cycle for those unique states, when they are needed in *Dijkstra's* shortest path algorithm to find the shortest error trace.

To the best of our knowledge, techniques to generate counterexamples in symbolic model checking done by Clarke et al. [4] and Jin et al. [9] are the closest related work to ours. Their approaches can only be used in symbolic model checking, while our approach can be used for any error trace generated by simulation or formal verification tools. In general,

*This work was supported by the National Science Council, R. O.C., under contract no. NSC89-2215-E009-120, while both authors were at National Chiao-Tung University in Taiwan.

test vector generation has been the subject of many efforts in testing and function verification areas. For instance, formal verification-based techniques to derive a set of function test vectors for simulation have been reported by Benjamin et al. [2], Geist et al. [6], Gupta et al. [7] and Ho et al. [8]. However, these approaches are generating test vectors for detecting bugs in designs. Our work is to generate shorter test vectors from the initial error trace for debugging process.

The rest of this paper is organized as follows: In section II, the definition of the problem is described. Section III describes our approach to solve the problem. The experimental results are shown in section IV. Section V describes our conclusions and future work.

II. Problem Definition

A digital circuit can be formulated as a finite state machine (FSM) and the circuit behavior can be viewed as the sequences of state traversal. The error trace contains all information of a sequence of state traversal from an initial state to an error state. First, we observe some characteristics of the error trace. Figure 1 shows the state transition diagram of a simple finite state machine and an initial error trace. The error trace takes 9 cycle time to reach the target error state. However, notice that three of the passed states, $state$ 1, 2, and 3, are traversed twice. In fact, it just passes through 6 unique states totally, and hence the shortest trace takes 6 cycle time at most to reach the error state in the worst case. If state 5 rather state 4 was chosen as the next state when the state 3 was traversed first time, the duplicate state traversal can be avoid. However, we can't know which next state should be chosen to avoid the duplicate state traversal until the whole error trace is traversed. It is a quite popular circumstance to have many redundant states in the error traces generated by random/pseudo-random simulation. Thus, we want to find a shorter trace to reach the same target error state from the same initial state.

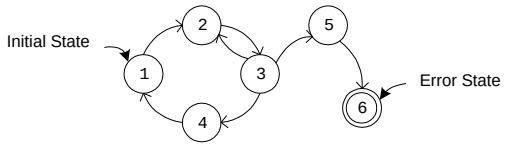

Fig. 1. The state transition diagram of a simple FSM and an initial error trace, where state 1, 2 and 3 are traversed twice.

The problem can be defined as follows: Given a circuit design and its error trace containing the initial state and the error state, the problem is to find the shortest error trace from the initial state to the error state and then to generate the corresponding input/output test vectors of the new trace. In addition, we have to make the following assumptions on the input designs:

1. Completely synchronous digital circuits.
2. Global reset signal always available.
3. no 3-state latches or registers.

III. Our Approach

This section presents our approach for compacting the error trace. The basic idea of our approach is based on the concept of eliminating the redundant states in the error trace and finding the shortest path among the remaining unique states in the trace. Figure 2 shows *CET1* algorithm for compacting the error trace. First, redundant states on the initial error trace are eliminated by the function Unique() to generate unique states US, initial state IS and error state ES. Then , the transition functions TF = $\delta(s,x)$ and output functions OF= $\lambda(s,x)$ of the circuit C are build by function BuildFunc() and are represented by a set of Binary Decision Diagrams (BDD) [3], where s is the set of current state BDD variables and x is the set of input BDD variables. The graph G is initialized with all states in US as the vertexes and no edges among them. For each state u in the unique states US, The reachable states by one cycle are computed by function ComupteNS() and the edges with weight 1 between u and the reached next states are added into graph G. *Dijkstra's* shortest path algorithm is applied to find the shortest path $SSeq$ of graph G from the initial state IS to the error state ES.

Algorithm: CET1(C,Seq) {
Output: IO—Test patterns for new error trace.
1 US, IS, ES = Unique(Seq);
2 TF, OF = BuildFunc(C);
3 G contains all states in US without any edges;
4 For each state u in US
5 NS= ComputeNS(TF,u,US);
6 Add edges with weight 1 between s and states in NS;
7 $SSeq$ = DijkstraSP(G, IS, ES);
8 IO = GenIO($TF, OF, SSeq$);
9 }

Fig. 2. *CET1* algorithm for compacting error trace.

Finally, according to the shortest path, input/output test patterns are generated by function GenIO(). The following tasks are performed for every state transition $< u, v >$ in the new trace: generating the input vectors and generating the corresponding output vectors. The input vector for state transition from state u to state v can be computed by solving $I(x)$=$\exists\, t\ \exists\, s\ u(s) \cdot v(t) \cdot (t \equiv \delta(s,x))$, where t is the set of next state BDD variables, $u(s)$ is the BDDs representing state u and $v(t)$ is the BDDs representing state v. Since $I(x)$ represents all possible input patterns, we random choose one from $I(x)$. Given the state u and the input vector iv, the corresponding output vector can be computed by solving OF=$\lambda(s,x)$ by substituting s with u and x with iv.

A. Finding Unique States in Initial Error Trace

In order to quickly find out the duplicate state in the initial error trace, we build one data structure,called *non-duplicate tree*, as shown in Figure 3, based on the assumption that each latch or register just could contains one of the two values, 0 or 1, at any time. The left branch represents encoding 0, the right one represents encoding 1, and the height of the tree represents the number of state bits. While each path from the root node to each leaf node represents one state appears on

the error trace. During the *non-duplicate tree* construction, if the traversed path according to state encoding has already existed, we can affirm the state has been reached previously, and then ignore it to deal with the next state continuously. Otherwise, it means the state has not been reached previously, and we should add some required nodes and branches to construct the *non-duplicate tree*.

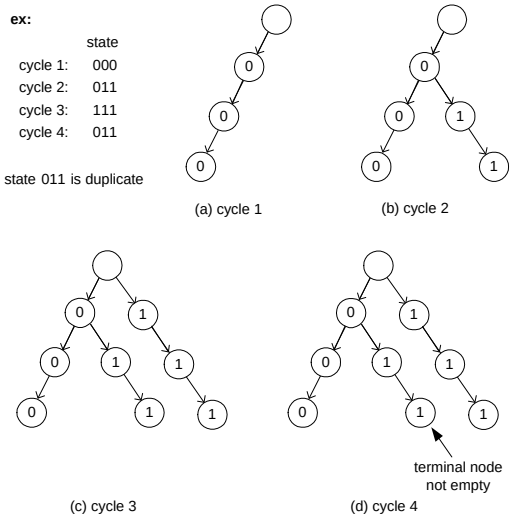

Fig. 3. Steps to build *non-duplicate tree.*

This *non-duplicate tree* is represented by Algebraic Decision Diagrams (ADD) [1] where the leaf nodes contain the unique state identification numbers. BDDs are generated to represent these unique states. Moreover, the length of the shortest error trace will not longer than the number of unique states, since each state on the shortest error trace should be traversed at most once. What is the time complexity of this method? With **N** states on the error trace and **b** state bits, the time complexity is $\boldsymbol{O(N \times b)}$.

B. Computing 1-cycle reachable states

Figure 4 shows the algorithm for computing 1-cycle next states for a given state u and the unique states US. The image computation operation $Image()$, popularly used in symbolic model checking [10], is used to compute the one-cycle reachable states N of state u under the set of transition functions TF. With image computation, the whole reached next states from state u can be found. Since we are only interested in the states in the unique states US, we then perform a BDD "And" operation on states N and unique states US to obtain the reached unique states R.

Algorithm: ComputeNS (TF, u, US){
Output: *Reached*—the reached next states.
1 $N = Image(TF, u)$;
2 $R = US \cap N$;
3 }

Fig. 4. The algorithm for computing 1-cycle reachable states.

C. Finding the Shortest Path

After building the connected graph G with weighted edges, the task is to find the shortest error trace from initial state IS to the target error state ES. *Dijkstra's* shortest path algorithm [5], shown in Figure 5, is a very suitable approach for our requirement. First, the graph G is initialized to have infinite distance for each state, except that the distance of the initial state IS is set to 0. Then, state IS is put into the priority queue Q keyed by their distance values. The algorithm repeats the following processes until ES is selected from Q. First, state u is extracted from Q with the smallest distance value by function ExtractMin(). Then, for each state v is connected to u, the function Relax() consists of testing whether we can improve the shortest path to v found by going through u and if so, updating the new distance of v and its predecessor to u. Finally, the shortest path is formed by tracing its predecessor of the states starting from ES to IS.

Algorithm: DijkstraSP(G, IS, ES){
Output: $path$—the shortest path from IS to ES.
1 Initialize(G,IS);
2 Q=IS
3 while ((u= ExtractMin(Q)) is not equal to ES)
4 for each state v is connected to u
5 Relax(u,v,G);
6 $path$ = findPath(G,IS,ES);
7 }

Fig. 5. Modified Dijkstra Shortest Path algorithm.

D. CET2 Algorithm

Algorithm: CET2(C,Seq) {
Output: IO—Test patterns for new error trace.
1 US, IS, ES = Unique(Seq);
2 TF, OF = BuildFunc(C);
3 G contains all states in US without any edges;
4 Initialize(G,IS);
5 Q=IS
6 while ((u= ExtractMin(Q)) is not equal to ES)
7 NS= ComputeNS(TF,u,US);
8 for each state v in NS
9 Add edge $< u, v >$ with weight 1;
10 Relax(u,v,G);
11 $SSeq$ = findPath(G,IS,ES);
12 IO = GenIO($TF, OF, SSeq$);
13 }

Fig. 6. *CET2* algorithm for compacting error trace.

We found that most of the time for algorithm *CET1* is spent on computing the 1-cycle reachable states for each unique state and these reachable states information is needed to perform function Relax() in Figure 5. If we can avoid performing function ComputeNS() as many as possible, our algorithm can have

better performance. Thus, we reschedule the time to perform function ComputeNS() into *Dijkstra's* shortest path algorithm, as shown in Figure 6. Algorithm *CET2* rearranges the tasks in line 4-7 of algorithm *CET1* into the new order shown in line 4-10 of Figure 6. With this modification, function ComputeNS() will not be performed for those states which shortest distance from IS is greater than the shortest distance from IS to ES, since they will not be extracted from Q. In worst case, *CET2* will have the same performance as *CET1*.

IV. Experimental Results

We have implemented our *CET1* and *CET2* algorithms in C++ language with the CUDD (Colorado University Decision Diagram) package [11]. We applied our algorithms on twelve designs in ITC'99 benchmarks. The circuit information about those designs is given in Table 1. The first and second columns list the number of circuits and the original functionality respectively. The third column is the total number of gates in the designs. The numbers of the primary inputs and outputs are given in the fourth and fifth columns respectively. The last column is the total number of registers in the designs. For each of these designs, we generated several error traces with different lengths and our algorithms on them. The experiments were performed on a 1.4 GHz AMD Athlon machine with 2GB main memory.

Circuit	Original Functionality	Gates	PI	PO	FF
$b1$	FSM that compares serial flows	47	3	2	5
$b2$	FSM that recognizes BCD numbers	29	2	1	4
$b3$	Resource arbiter	150	5	4	30
$b4$	Compute min and max	606	12	8	66
$b5$	Elaborate the contents of memory	977	2	36	34
$b6$	Interrupt handler	61	3	6	9
$b7$	Count points on a straight line	422	2	8	49
$b8$	Find inclusions in sequences of numbers	168	10	4	21
$b9$	Serial to serial converter	160	2	1	28
$b10$	Voting system	190	12	6	17
$b11$	scramble string with variable cipher	484	8	6	31
$b12$	Interface to meter sensors	343	11	10	53

TABLE 1 Design information

Table 2 shows the experimental results using *CET1* algorithm. The circuit name and trace number is shown in column 1. The number of states of the initial error trace is shown in column 2. Column 3 shows the number of unique states in the initial error trace. In *CET1*, the number of unique states in the trace is the number of states needed to compute their reachable next states, and is the upper bound of the final result. From the results, case "b1-1" has the lowest percentage (7.3%) and several cases have no redundant states in the traces. Column 4 shows the number of states in the shortest error trace found by *CET1* algorithm. For most of the cases, the length of the error trace is reduced dramatically. For instance, the length of error trace is reduced from 40061 to 17 in case "b8-1". For cases in circuit "b4", *CET1* can found the shortest error traces within 6% of the length of their initial traces, which have almost no redundant state. However, for case "b5-2" and "b12-1", the length of their error traces can not be reduced at all. For case "b5-1", the length of the error trace is reduced by only 1 state. CPU time and memory usage of *CET1* algorithm, reported in columns 5 and 6, respectively, is proportional to the number of unique states shown in column 3, because for each unique state, *CET1* compute its reachable next states to build the graph and then apply *Dijkstra's* shortest path algorithm to find the shortest error trace. For most cases, except cases in circuit "b4", *CET1* can generate the shortest error trace within 300 seconds and 100 MB memory usage. For cases in circuit "b4", we found most of the CPU time and memory of *CET1* spent on computing the reachable next states.

Circuit -Trace#	States			CPU time (sec)	Memory (MB)
	Init	Unique	Final		
b1-1	233	18	4	0.15	0.08
b1-2	73	18	4	0.14	0.07
b2-1	14	7	5	0.13	0.06
b2-2	56	8	4	0.10	0.06
b3-1	20493	1822	27	3.76	2.96
b3-2	32352	1936	20	3.98	3.62
b4-1	3037	3037	168	2847.40	12.28
b4-2	5123	5123	232	4882.02	17.47
b4-3	7641	7641	227	6864.21	24.11
b4-4	9380	9379	167	8180.27	27.88
b4-5	9719	9718	200	8937.08	29.12
b5-1	111	111	110	0.55	1.02
b5-2	106	106	106	0.59	1.04
b6-1	85	13	5	0.16	0.09
b6-2	59	13	4	0.12	0.09
b7-1	129	128	88	1.68	3.02
b7-2	167	87	83	1.45	3.10
b8-1	40062	15660	19	23.57	13.97
b8-2	20246	11314	28	22.70	10.50
b9-1	16207	12384	50	31.81	14.54
b9-2	40355	26448	46	102.83	27.78
b10-1	13924	2053	19	2.45	1.89
b10-2	15203	2071	20	2.55	1.95
b11-1	28738	23936	275	169.88	45.09
b11-2	56650	43370	80	272.43	92.18
b12-1	812	812	812	5.08	1.87
b12-1	189	81	51	0.82	0.84

TABLE 2 The experimental results of *CET1* algorithm.

Table 3 shows the CPU time and memory usage of *CET1* and *CET2* algorithms for cases in circuit "b4" and case "b11-2", which requires longer CPU times to find the shortest error trace. For other cases, *CET2* has the similar CPU time and memory usage as *CET1* shown in Table 2. *CET2* found the same length of shortest error trace as *CET1* did. In general, *CET2* used less memory usage to achieve the same quality as *CET1* did, while *CET2* improve the CPU time dramatically for

cases in "b4". The CPU time improvement is contributed by that the number of states to compute the next states in *CET2* is reduced from the number of unique states to the number states with length not greater than the length of the shortest error trace. For instance, in case "b4-4", *CET1* compute the reachable next states for 9380 states, *CET2* only compute the reachable next states for states with length less than 167 from initial state. Thus, *CET2* has 37 times speedup than *CET1* for this case.

Circuit	CPU Time (sec)			Memory (MB)	
-Trace#	*CET1*	*CET2*	speedup	*CET1*	*CET2*
b4-1	2847.40	183.58	15.5	12.28	11.79
b4-2	4882.02	933.54	5.2	17.47	16.65
b4-3	6864.21	1005.14	6.8	24.11	22.92
b4-4	8180.27	221.32	37.0	27.88	26.37
b4-5	8937.08	558.03	16.0	29.12	27.56
b11-2	272.43	153.58	1.8	92.18	51.38

TABLE 3 The experimental results of *CET1* and *CET2* algorithms.

V. Conclusions and Future Work

We have shown the concept of compacting the error traces generated by pseudo-random/random simulations and two algorithms *CET1* and *CET2* to perform the task of compacting the error trace. Two algorithms *CET1* and *CET2* are presented to solve this problem efficiently. Experimental results show that both algorithms can reduce the length of error traces dramatically for most cases using reasonable memory. For some cases, our approach can not reduce the length of error traces, since they are the shortest one already. For cases required longer CPU time to find the shortest trace, *CET2* is up to 37 times faster than *CET1*.

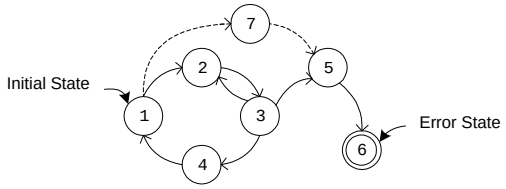

Fig. 7. If the error trace is able to pass through $state$ 7, there exists one shorter error trace, 1→7→ 5→6, which takes 3 cycles.

However, there are still many ways to improve our algorithms. First, we would like to explore other heuristics to speed up our algorithms by reducing the number states to compute the reachable next states. Second, we would like to reduce the length of the shortest error trace as shorter as possible. Currently, we only compute one-step successors in our algorithms. In some case, we can not reduce the length of the error trace. Thus, one-step successors may not be enough to achieve shorter trace. For examples, in Figure 7, State 7 is one of the next states of state 1, but is not one of unique states. Thus, our algorithms doesn't include state 7 in the reachable states. However, there exists one path passing through state 7 to reach the target error state in 3 cycles, and our algorithm can only find the path with length 4 cycles. In order to find shorter error trace, we plan to add the length parameter n into function ComputeNS() to find n-cycle successors. Thus, edges among unique states may contains different edge weights from 1 to n. With this modification, we will be able to find shorter error trace. However, the more cycles we compute the reachable states, the shorter error trace we can find and the slower performance we have to suffer. Thus, n can not be too large.

References

[1] I. Bahar, E. Frohm, C. Gaona, G. Hachtel, E. Macii, A. Pardo and F. Somenzi, "Algebraic Decision Diagrams and their Applications" In Journal of Formal Methods in System Design, Volume 10, Number 2/3, April/May 1997.

[2] Mike Benjamin, Daniel Geist, Alan Hartman, Yaron Wolfsthal, Gerard Mas, and Ralph Smeets, "A Study in Coverage-Driven Test Generation", 36^{th} Design Automation Conference, June 1999, pp. 970-975.

[3] R. E. Bryant. "Graph-based algorithms for Boolean function manipulation", IEEE Transaction Computers, C-35(8):677-691, August 1986.

[4] E. M. Clarke, O. Grumberg, K.L. McMillan and X. Zhao, "Efficient Generation of Counterexamples and Witnesses in Symbolic Model Checking", in 32^{nd} Design Automation Conference, June 1995, pp. 427-432.

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, " Introduction to Algorithms", McGraw-Hill Book Company, pp. 527-531.

[6] D. Geist, M. Farkas, A. Landver, Y. Lichtenstein, S. Ur, and Y. Wolfsthal, "Coverage Directed Test Generation Using Symbolic Techniques", In Formal Methods in Computer-Aided Design, Nov. 1996, pp. 143-158.

[7] Aarti Gupta, Sharad Malik, and Pranav Ashar, "Toward Formalizing a Validation Methodology Using Simulation Coverage", 34^{th} Design Automation Conference, June 1997, pp. 740-745.

[8] P.-H. Ho, T. Shiple, K. Harer, J. Kukula, R. Damiano, V. Bertacco, J. Taylor and J. Long, "Smart Simulation Using Collaborative Formal and Simulation Engines", In International Conference on Computer Aided Design, Nov. 2000, pp. .

[9] H. Jin, K. Ravi and F. Somenzi, "Fate and Free Will in Error Traces", in 2002 Conference of Tools and Algorithms for the Construction and Analysis of Systems, 2002, pp. 445-459.

[10] K. L. McMillan, " Symbolic Model Checking", Kluwer Academic Publishers, Boston, MA, 1994.

[11] F. Somenzi, "CUDD: CU decision diagram package - Release 1.0.4", Tech, Rep., Dept. Elect. Comput. Eng., Univ. Colorado, Boulder, Nov. 1995.