An Optimum Algorithm for Compacting Error Traces for Efficient Design Error Debugging

Chia-Chih Yen and Jing-Yang Jou, *Fellow*, *IEEE*

Abstract—Diagnosing counterexamples with error traces has acted as one of the most critical steps in functional verification. Unfortunately, error traces are normally very lengthy such that designers need to spend considerable effort to understand them. To alleviate the designers' burden for debugging, we present a SAT-based algorithm for reducing the lengths of error traces. The algorithm performs the paradigm of the binary search algorithm to halve the search space recursively. Furthermore, it applies a novel theorem to guarantee gaining the shortest lengths for the error traces. Based on the optimum algorithm, we develop two robust heuristics to handle real designs. Experimental results demonstrate that our approaches greatly surpass previous work and, indeed, have promising solutions.

Index Terms—Verification, simulation, diagnosis, error checking, satisfiability.

1 Introduction

Employing assertions [1], [2] to ensure functional correctness on hardware designs has increasingly become a dominant methodology to surmount today's verification obstacles. Assertion-based verification (ABV) benefits not only from enhancing design observability, but also from reducing debugging time. In general, the ABV process is comprised of three phases: writing assertions, detecting violation of assertions, and debugging errors. To efficiently discover the counterexamples of the assertions, ABV operates different kinds of verification technologies to collaborate. As a rule, these technologies include random/pseudorandom simulation, symbolic fixed-point computation [3], and SAT-based bounded model checking [4].

Once the verification engines detect a counterexample, designers need to diagnose it to find the causes of the error. In ABV, a counterexample is simply an error trace that lists a set of states in a design. Such a set of states forms a specific path that ends at a state violating an assertion. Ideally, designers can quickly locate the faulty portions of the design by simulating the error trace and viewing the waveform. In practice, however, an error trace may be fairly lengthy such that understanding a counterexample continues to be a very difficult task, usually requiring considerable effort.

In this paper, we present an algorithm that is intended to reduce the lengths of error traces. The benefits for compacting error traces are twofold. First, since human effort is, so far, the chief approach to debugging counterexamples, we believe that the compact error traces will greatly assist designers in reasoning through the errors easily. Second, after compacting an error trace, designers may obtain another different counterexample for the same assertion. Through the additional counterexample, designers may thus learn more information about the bugs and thus improve the efficiency for diagnosing functional errors.

1.1 Problem Formulation

Given a synchronous sequential design with a global reset signal, we define a state of the design as a combination of the values of all sequential elements. Theoretically, each state contains specific information about the design.

Assume an initial state in a design is the state while the reset signal is active. Then, we define an error trace as a sequence of distinct states starting at the initial state and ending at an error state, where the error state is a state that has some properties violating an assertion. The length of an error trace is the number of edges from the initial state to the error state. Since the states in an error trace are distinct, the length of an error trace is simply the number of states minus one. Given an original error trace with the length n, the goal of the error trace compaction problem is to find another error trace with the same error state and having a shorter length $n' < n$.

Fig. 1 illustrates an example of error traces. Fig. 1a pictures the state transition diagram, and Fig. 1b shows that the original error trace takes 13 cycles in Fig. 1a, that is, $n = 13$, from the initial state S^0 to the error state S^{14}. Clearly, based on Fig. 1a, we can identify several compact error traces whose lengths are smaller than 13. Fig. 1c and Fig. 1d show two examples of compact error traces, where Fig. 1d is the shortest one in the design.

1.2 Previous Work

Previous work has seldom addressed the error trace compaction problem. In the VLSI testing area, many proposed approaches focus on compacting test vectors [5], [6]. They attempt to compact the lengths of test vectors while keeping the fault coverage unchanged. Nevertheless, the major intention of the error trace compaction problem is to reduce the lengths of the error traces while preserving the

- *The authors are with the Department of Electronics Engineering, National Chiao-Tung University, 1001 Ta-Hsueh Road, Hsinchu, Taiwan 30050, ROC. E-mail: jackr@eda.ee.nctu.edu.tw, jyjou@faculty.nctu.edu.tw.*

Manuscript received 29 Aug. 2005; accepted 24 Feb. 2006; published online 21 Sept. 2006.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TCSI-0289-0805.

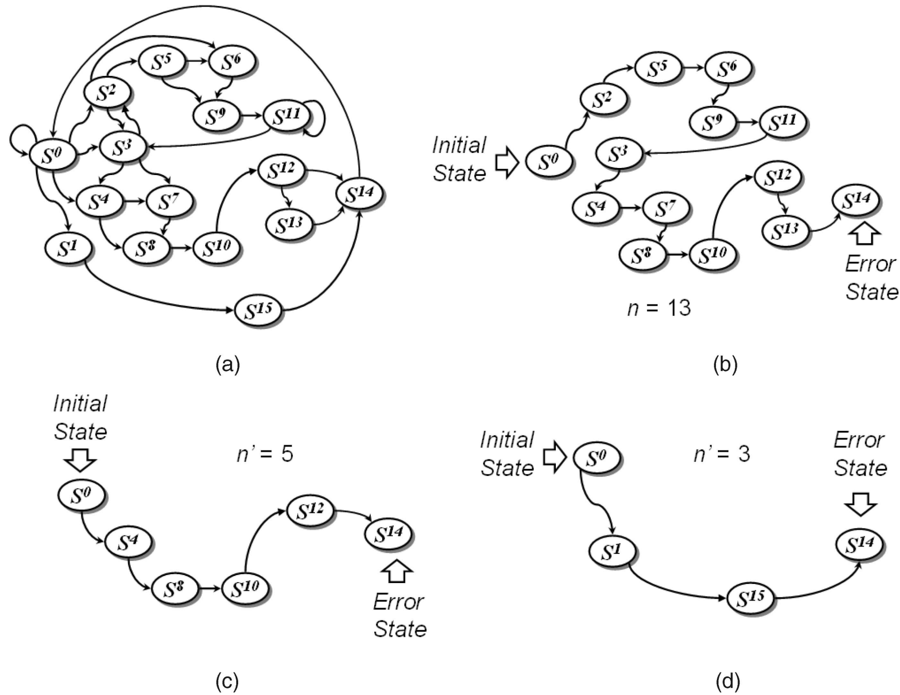


Fig. 1. An example of an original error trace and its compact error traces: (a) a state transition graph, (b) the original error trace, (c) a compact error trace, and (d) another compact error trace with the shortest length.

same error states. Hence, the work of test compaction is orthogonal to our goal.

In [7], Chen and Chen present two algorithms, named *CET1* and *CET2*, for compacting error traces. They suppose that the counterexamples are all generated by random/pseudorandom simulation and, thus, the error traces must consist of many redundant states. Based on such assumption, *CET1* and *CET2* first attempt to find unique states in an original error trace. Then, *CET1* begins to build a connected graph among those unique states. To fulfill this manipulation, it applies BDDs-based one-cycle image computation to each state. After the connected graph is ready, *CET1* then performs *Dijkstra's* shortest path algorithm [8] to obtain the shortest error trace in the graph. Similarly to *CET1*, *CET2* still uses *Dijkstra's* algorithm. However, *CET2* embeds the procedure of building connected graph into *Dijkstra's* approach. Such a step benefits *CET2* by avoiding building unnecessary state connections. Consequently, the lengths of compact error traces are the same in both *CET1* and *CET2*, while *CET2* gains more runtime efficiency.

The major flaw in CET1 and CET2 is that they only apply limited distinct states to create the connected graph and, therefore, the solution space is confined to those states. Specifically, the results obtained by *CET1* or *CET2* are only the local optimum solutions. If the unique states in the original error trace are insufficient, then both algorithms may possibly gain no improvement after the compaction.

1.3 Our Approach

Instead of identifying the shortest path among the limited distinct states, we present a SAT-based algorithm that takes the binary search paradigm to find the global optimum solution. In general, a binary search oriented algorithm contains two traits: First, the algorithm must have a specific partitioning criterion such that it can halve the solution space at each recursion; second, the algorithm must have a particular termination condition to end the search. In our approach, we apply two theorems to meet both requirements. For the partitioning criterion, Theorem 1 provides a hint to check if the length of the shortest path for an error trace is larger or smaller than the half of the original length. Based on this theorem, our algorithm can eliminate half of the solution space and then search the rest iteratively. For the termination condition, Theorem 2 elaborates on how to determine the shortest path between two distinct states in a design. With the theorem, our algorithm can cease the binary checking process and then obtain the shortest length for the error state.

By elegantly utilizing Theorem 1 and Theorem 2, our algorithm not only compacts error traces effectively but also identifies the optimum solutions. Through the manipulation of the method, designers may earn numerous profits to increase their debugging performance.

1.4 Paper Organization

The remainder of this paper is organized as follows: We state preliminaries in Section 2. In Section 3, we elaborate two theorems and then present our optimum algorithm. To demonstrate the effect of our approach, we present experimental results in Section 4. Moreover, we develop two robust heuristics for handling real designs in Section 5. Finally, we give the conclusions and future work in Section 6.

2 Preliminaries

We model a synchronous sequential design as a finite state machine (FSM) M. Assume each M has a reset input that can transfer any states in M into the initial state. The state

transition graph of an FSM, $STG(M)$, is a directed graph (V, E), where each vertex $v \in V$ corresponds to a state in M and each edge $e \in E$ corresponds to a state transition between two states. A self-transition state is a state that has an edge transfer from the state to itself. Apparently, the initial state in M is a self-transition state.

Given an $STG(M)$, we define the following two functions that will be used in our algorithm. We implement both procedures by the SAT-solvers.

Definition 1. *The procedure* $\mathrm{WALK}(k, u, v)$ *is a query function, where* k *is a natural number, and* $u, v \in V$ *in the* $STG(M)$. $\mathrm{WALK}(k, u, v)$ *is true when there exists a succession of* k *directed edges in* $STG(M)$*, starting at vertex* u *and ending at vertex* v*. Otherwise, it is false.*

Definition 2. *The procedure* $\mathrm{PATH}(k, u, v)$ *is a query function, where* k *is a natural number, and* $u, v \in V$ *in the* $STG(M)$. $\mathrm{PATH}(k, u, v)$ *is true when there exists a succession of* k *directed edges in* $STG(M)$*, starting at vertex* u *and ending at vertex* v*, and all vertices on these edges are distinct. Otherwise, it is false.*

As an example, consider the $STG(M)$ depicted in Fig. 1a. The function $\mathrm{WALK}(4, S^0, S^2)$ is true because the following state transition exists: $S^0 \rightarrow S^0 \rightarrow S^0 \rightarrow S^0 \rightarrow S^2$. However, $\mathrm{PATH}(4, S^0, S^2)$ is false since there do not exist four distinct states from S^0 to S^2. On the other hand, both functions $\mathrm{WALK}(3, S^0, S^{14})$ and $\mathrm{PATH}(3, S^0, S^{14})$ are true due to the transition $S^0 \rightarrow S^1 \rightarrow S^{15} \rightarrow S^{14}$.

3 Optimum Error Trace Compaction Algorithm

In this section, we detail how we come up with our error trace compaction algorithm. Since the algorithm attempts to apply the binary search paradigm, we first present two theorems to satisfy the requirements for the paradigm. Then, we give the entire pseudocode for our algorithm and prove the reason why the algorithm acquires the optimum solution.

3.1 Halving the Solution Space

Given an original error trace with the length n, are there any methods to know if the length of the shortest path from the initial state to the error state is either between 1 and $n/2$ or between $((n/2) + 1)$ and n? Theorem 1 [9] provides a hint on the question.

Theorem 1. *Given an* $STG(M)$ *with two vertices* $u, v \in V$*. If* u *is a self-transition state, then the following equation holds:*

$$\mathrm{WALK}(k, u, v) \leftrightarrow \mathrm{PATH}(1, u, v) \cup \mathrm{PATH}(2, u, v) \cup \ldots \cup \mathrm{PATH}(k, u, v).$$

Correspondingly, the negation of both sides also holds:

$$\neg\mathrm{WALK}(k, u, v) \leftrightarrow \neg\mathrm{PATH}(1, u, v) \cap \neg\mathrm{PATH}(2, u, v) \cap \ldots \cap \neg\mathrm{PATH}(k, u, v).$$

Proof. Based on the basic graph theory, we have

$$\mathrm{WALK}(k, u, v) \rightarrow \mathrm{PATH}(1, u, v) \cup \mathrm{PATH}(2, u, v) \cup \ldots \cup \mathrm{PATH}(k, u, v). \quad (1)$$

On the other hand, assume $\mathrm{PATH}(m, u, v)$ is true, where m is between 1 and k; then, we can create a transition of distance m from u to v. Moreover, since u is a self-transition state, we can create a transition of distance $(k - m)$ from u to itself. Next, if we concatenate the above two transitions, we can create a transition of distance $(k - m) + m = k$ from u to v and such a transition suggests that $\mathrm{WALK}(k, u, v)$ is true by definition. In short, if $\mathrm{PATH}(m, u, v) = \mathrm{true}$, where $1 \leq m \leq k$, then $\mathrm{WALK}(k, u, v) = \mathrm{true}$. Based on the above deduction, we have the following set of equations:

$$\begin{aligned} &\mathrm{PATH}(1, u, v) \rightarrow \mathrm{WALK}(k, u, v) \\ &\mathrm{PATH}(2, u, v) \rightarrow \mathrm{WALK}(k, u, v) \\ &\ldots \\ &\mathrm{PATH}(k, u, v) \rightarrow \mathrm{WALK}(k, u, v). \end{aligned}$$

Consequently, we acquire the following formula:

$$\mathrm{PATH}(1, u, v) \cup \mathrm{PATH}(2, u, v) \cup \ldots \cup \mathrm{PATH}(k, u, v) \rightarrow \mathrm{WALK}(k, u, v). \quad (2)$$

According to (1) and (2), we prove Theorem 1. □

Theorem 1 suggests that the function $\mathrm{WALK}(k, u, v)$ decides whether the length of the shortest path from u to v is larger or smaller than k. Such a decision ability enables us to discard half of the solution space for the error trace compaction problem. The following elaborates on how it works.

Consider an error trace with the initial state u, the error state v, and the original length n. Apparently, the length n for the path from u to v implies the following equation:

$$\mathrm{WALK}(n, u, v) = \mathrm{true}.$$

According to Theorem 1, the above equation implies that at least one of the following equations holds ($1 \leq k \leq n$):

$$\left.\begin{aligned} &\mathrm{PATH}(1, u, v) = \mathrm{true} \\ &\mathrm{PATH}(2, u, v) = \mathrm{true} \\ &\ldots \\ &\mathrm{PATH}(k, u, v) = \mathrm{true} \end{aligned}\right\}(\mathrm{I})$$
$$\left.\begin{aligned} &\mathrm{PATH}(k+1, u, v) = \mathrm{true} \\ &\ldots \\ &\mathrm{PATH}(n-1, u, v) = \mathrm{true} \\ &\mathrm{PATH}(n, u, v) = \mathrm{true} \end{aligned}\right\}(\mathrm{II}).$$

Obviously, to find the shortest path from u to v, we can directly identify the above PATH functions from 1 to n sequentially. However, such a brute force method is very inefficient. Therefore, we intend to apply the binary search manner to discover which PATH function is true and has the smallest length.

To realize the location for the length of the shortest path from u to v, we divide the above equations into two groups.

Group (I) stands for the path of length between 1 and k and group (II) represents the path of length between $(k+1)$ and n.

Next, we attempt to halve the solution space by checking the function $\mathrm{WALK}(k,u,v)$. If $\mathrm{WALK}(k,u,v)$ is true, Theorem 1 ensures that at least one of the equations in group (I) holds. That is, the length of the shortest path from u to v is guaranteed to be between 1 and k. To the contrary, in case $\mathrm{WALK}(k,u,v)$ is false, then none of the equations in group (I) holds. However, recall that the original length n from u to v suggests that at least one of the equations in (I) and (II) holds. Thus, we comprehend that at least one of the equations in group (II) holds. In other words, the length of the shortest path from u to v is guaranteed to be between $(k+1)$ and n.

Apparently, if we let k be the midpoint of n, $k = n/2$, then, through the use of checking $\mathrm{WALK}(n/2,u,v)$, we narrow down the range for the location of the length of the shortest path from u to v. Take Fig. 1b as an example. Since the original length $n = 13$, we can check $\mathrm{WALK}(13/2, S^0, S^{14})$ or $\mathrm{WALK}(6, S^0, S^{14})$ to decide the location for the length of the shortest path. Based on Fig. 1a, the answer to the function is true; therefore, we realize that the length of the shortest path from S^0 to S^{14} is between 1 and 6.

3.2 Terminating the Search

By recursively checking the WALK function, we lessen the solution space step by step. However, we still need to know when to terminate the searching process to determine the actual shortest path. Theorem 2 states such a condition.

Theorem 2. *Given an $STG(M)$ with two vertices $u, v \in V$, where u is a self-transition state. If the following two equations hold:*

$$\mathrm{WALK}(k-1,u,v) = \mathrm{false}$$
$$\mathrm{WALK}(k,u,v) = \mathrm{true},$$

then the length of the shortest path from u to v is exactly k.

Proof. Based on Theorem 1, if $\mathrm{WALK}(k,u,v)$ is true, then at least one of the following functions: $\mathrm{PATH}(1,u,v)$, $\mathrm{PATH}(2,u,v)$, $\ldots$, $\mathrm{PATH}(k-1,u,v)$, or $\mathrm{PATH}(k,u,v)$ is true. Nevertheless, the equation $\mathrm{WALK}(k-1,u,v) = \mathrm{false}$ means that all of the following functions: $\mathrm{PATH}(1,u,v), \mathrm{PATH}(2,u,v), \ldots,$ and $\mathrm{PATH}(k-1,u,v)$ are false. Thus, if both equations $\mathrm{WALK}(k-1,u,v) = \mathrm{false}$ and $\mathrm{WALK}(k,u,v) = \mathrm{true}$ occur at the same time, we infer that the equation $\mathrm{PATH}(k,u,v) = \mathrm{true}$ is the only factor to cause $\mathrm{WALK}(k,u,v)$ to be true. It implies that the length of the shortest path from u to v is exactly k. □

Theorem 2 clearly describes the condition for determining the shortest distance of an error trace between the initial state u and the error state v. With the support of Theorem 1 and Theorem 2, we develop our optimum error trace compaction algorithm in the following section.

```
// IS: the initial state
// ES: the error state
// n: the original length of the error trace
// n': the final length of the error trace
01  OETC-BIN-SEARCH ( n , IS , ES ) {
02    LB = 0;
03    UB = n;
04    while ( (UB - LB) != 1 ) {
05       MP = (LB + UB)/2 ;
06       if ( WALK (MP, IS, ES) == true )
07           UB = MP ;
08       else
09           LB = MP ;
10    }
11    n' = UB ;
12    PRINT-PATH ( n' , IS , ES ) ;
13    return n' ;
14  }
```

Fig. 2. The binary search algorithm for compacting error traces.

3.3 The Optimum Algorithm

Fig. 2 depicts the pseudocode of our error trace compaction algorithm. It accepts three inputs: the initial state IS, the error state ES, and the original length n. First, the algorithm assigns the values for two variables, LB and UB, which, respectively, represent the lower bound and the upper bound for the length of the shortest path from IS to ES. Afterward, the codes from line 4 to line 10 perform the key operation of the binary search. At each recursion, the algorithm sets the midpoint MP between LB and UB. Then, it checks $\mathrm{WALK}(MP, IS, ES)$ to halve the solution space. If the function is true, the shortest distance from IS to ES must be equal to or smaller than MP. Thus, the algorithm will replace UB with MP. To the contrary, if the function is false, the shortest distance from IS to ES must be larger than MP and, hence, the algorithm will replace LB with MP. Finally, if the difference between UB and LB is just one, the loop terminates and the algorithm prints the shortest path and returns UB as the shortest distance from IS to ES.

We explain the reason why the algorithm shown in Fig. 2 guarantees to acquire the optimum solution. For every UB at each recursion in line 7, $\mathrm{WALK}(UB, IS, ES) = \mathrm{true}$. Furthermore, for each LB in line 9, $\mathrm{WALK}(LB, IS, ES) = \mathrm{false}$. Whenever the termination condition in line 6 holds, $(UB - LB) = 1$, both of the following equations hold at the same time:

$$\mathrm{WALK}(LB, IS, ES) = \mathrm{false}$$
$$\mathrm{WALK}(UB, IS, ES) = \mathrm{true}.$$

Since LB is equal to $(UB-1)$, the above situation satisfies Theorem 2 and UB is absolutely the shortest distance from IS to ES.

We take the error trace example illustrated in Fig. 1b to demonstrate our algorithm. In the beginning, we set IS to S^0, ES to S^{14}, and n to 13. Then, we have the following recursions:

TABLE 1
Benchmark Information

CKT	# FF	# Gate	Sequential Depth
b10	17	206	-
b09	28	170	-
b03	30	160	7
b11	31	770	-
b04	66	737	8
s1269	37	569	10
s3271	116	1572	17
s3330	132	1789	9
s5378	179	2779	45
s6669	239	3080	-

Step 1: $LB = 0, UB = 13, \text{then } MP = (0 + 13)/2 = 6.$
$\text{WALK}(6, S^0, S^{14}) = \text{true, then reassign } UB = 6.$
Step 2: $LB = 0, UB = 6, \text{then } MP = (0 + 6)/2 = 3.$
$\text{WALK}(3, S^0, S^{14}) = \text{true, then reassign } UB = 3.$
Step 3: $LB = 0, UB = 3, \text{then } MP = (0 + 3)/2 = 1.$
$\text{WALK}(1, S^0, S^{14}) = \text{false, then reassign } LB = 1.$
Step 4: $LB = 1, UB = 3, \text{then } MP = (1 + 3)/2 = 2.$
$\text{WALK}(2, S^0, S^{14}) = \text{false, then reassign } LB = 2.$
Step 5: $LB = 2, UB = 3, \text{then } (UB - LB) = 1. \text{ Return } UB.$

After Step 5 finishes, we learn that the shortest distance from S^0 to S^{14} is $UB = 3$. Furthermore, through the SAT-solver, the algorithm outputs the shortest error trace that is just as Fig. 1d pictures.

On the whole, the benefits of the algorithm are twofold. First, it guarantees to obtain the optimum solution, that is, it identifies the shortest path from the initial state to the error state in the design. Second, given the original length n of an error trace, the algorithm performs the WALK function only $\log_2(n)$ times due to the binary search manner. Such a number of operation times gives a promising effect on the error trace compaction problem.

4 Experimental Results

To demonstrate the effectiveness of our algorithm, we implemented it in C++ and employed the Chaff [10] package as the underlying SAT-solver for the WALK function. Moreover, we also implement the *CET2* algorithm [7] presented in Section 1.2 for comparison. Note that, unlike the original *CET2* algorithm which employs BDDs, we implement it by the SAT-solver to handle large designs. Basically, such a change may influence the runtime. Nevertheless, the lengths of compact error traces are still the same with our implementation.

We conducted the previous work *CET2* and our optimum algorithm over some ITC-99 and ISCAS-89 benchmarks. All experiments were run on an AMD Athlon 64 3500+ workstation with 2GB main memory. Table 1 lists the fundamental information for these designs, where the first three columns display the names, the number of sequential elements, and the number of gates separately. The fourth column, titled "Sequential Depth," shows the sequential depths for every design. We use the approach presented in [11] with 5,000 seconds time limit to calculate the values, while the empty cells indicate that the sequential depth is not available within the limited runtime. In general, the sequential depth of a design means the largest length of all shortest paths starting from the initial state to any other reachable states. Hence, given any error traces, the shortest distances from the initial state to the error states must be equal to or smaller than the sequential depth. With the information of the sequential depths, we can prove that our algorithm certainly obtains the optimum solutions.

Table 2 presents the results for the error trace compaction. For each design, we generate three error traces by using the semiformal verification engine presented in [12]. Moreover, we preprocess all error traces such that the states on each error trace are distinct. The second and third columns in Table 2 give the names and the original lengths of the error traces. The fourth, fifth, and sixth columns show the results obtained by the *CET2* algorithm. Specifically, the fourth column presents the compact lengths of error traces and the fifth column shows the corresponding run time in seconds. To evaluate the effectiveness of the algorithm, the sixth column presents the percentage of reduction ratio, that is, the ratio of the reduced length to the original one. The higher the reduction ratio is, the better the operation the algorithm performs. Note that the 0 percent reduction ratio means the compact length is the same as the original one.

Similarly, the seventh, eighth, and ninth columns in Table 2 present the compact length, the runtime, and the reduction ratio obtained by our algorithm. The rightmost column, titled "IMP," gives the percentage of the improvement from the *CET2* algorithm to our algorithm. It calculates the difference from the values shown in the ninth column to the ones shown in the sixth column.

Based on Table 2, we see the following features for our algorithm. First, the compact results obtained by our algorithm are extremely superior to those by *CET2*. For example, for circuits b09, b04, and s6669, *CET2* could not reduce any lengths for the original error traces. On the contrary, our algorithm achieved at least 60 percent reduction for the three circuits. For b04, s3271, and s6669, our algorithm even improved over 90 percent reduction. The major reason for the great improvement is because our algorithm can globally search the entire state spaces, while *CET2* can just explore limited distinct states in the original error traces.

Second, compared to the sequential depths shown in the fourth column in Table 1, our algorithm indeed gained the shortest distance for each error trace. In other words, the results shown in the eighth column are all equal to or smaller than the sequential depths. As an example, b04 has the sequential depth 8 and the reduced lengths for b04-1, b04-2, and b04-3 are 8, 7, and 7, which certainly satisfies our deduction. Similarly, for b03, s1269, s3271, s3330, and s5378, the reduced results by our algorithm justify Theorem 2.

Third, our algorithm may become inefficient when handling complex circuits with very lengthy error traces. Take s5378-3 as an example. Although our algorithm acquired the shortest length 36, it took over 90,000 seconds.

TABLE 2
Experimental Results

CKT	Error Trace		CET2			Our Algorithm			IMP (%)
	Name	Original Length	Compact Length	Time (s)	Reduction (%)	Compact Length	Time (s)	Reduction (%)	
b10	b10-1	49	25	2.85	48.98	13	0.65	73.47	24.49
	b10-2	53	44	3.60	16.98	11	0.55	79.25	62.26
	b10-3	92	66	10.65	28.26	16	4.94	82.61	54.35
b09	b09-1	36	36	1.53	0	13	0.23	63.89	63.89
	b09-2	71	71	5.88	0	19	0.62	73.24	73.24
	b09-3	106	106	12.86	0	11	0.58	89.62	89.62
b03	b03-1	23	23	0.61	0	7	0.12	69.57	69.57
	b03-2	77	5	1.40	93.51	5	0.46	93.51	0.00
	b03-3	109	55	12.60	49.54	5	0.81	95.41	45.87
b11	b11-1	81	33	29.06	59.26	19	10.27	76.54	17.28
	b11-2	190	128	167.30	32.63	77	281.18	59.47	26.84
	b11-3	232	52	229.01	77.59	30	152.03	87.07	9.48
b04	b04-1	96	96	35.83	0	8	2.17	91.67	91.67
	b04-2	205	205	163.63	0	7	25.34	96.59	96.59
	b04-3	332	332	426.11	0	7	6.65	97.89	97.89
s1269	s1269-1	56	25	10.89	55.36	7	1.22	87.50	32.14
	s1269-2	83	9	16.36	89.16	6	2.74	92.77	3.61
	s1269-3	105	11	20.42	89.52	4	3.30	96.19	6.67
s3271	s3271-1	244	238	577.11	2.46	13	534.97	94.67	92.21
	s3271-2	406	404	1611.21	0.49	12	2082.96	97.04	96.55
	s3271-3	589	579	3381.54	1.70	12	1974.13	97.96	96.26
s3330	s3330-1	121	15	135.08	87.60	5	11.59	95.87	8.26
	s3330-2	173	25	269.94	85.55	7	24.54	95.95	10.40
	s3330-3	272	36	764.00	86.76	6	70.04	97.79	11.03
s5378	s5378-1	219	111	767.51	49.32	9	533.34	95.89	46.58
	s5378-2	381	90	2198.87	76.38	19	21509.60	95.01	18.64
	s5378-3	607	120	5763.71	80.23	36	95951.30	94.07	13.84
s6669	s6669-1	165	165	535.87	0	5	2354.94	96.97	96.97
	s6669-2	291	291	1671.79	0	5	2642.29	98.28	98.28
	s6669-3	422	422	3506.39	0	5	3293.98	98.82	98.82

Generally, such a flaw is due to the limit of the SAT-solvers. For example, since the original length of s5378-3 is 607, the first step in our algorithm is to solve $\mathrm{WALK}(607/2, IS, ES)$, or $\mathrm{WALK}(303, IS, ES)$. The length 303 in the WALK function means the SAT-solver has to solve 303 time-frame expansions for s5378, which is 303 times the gate count for the original circuit. Therefore, how to improve the efficiency while preserving promising results has become an important task.

5 Two Robust Heuristics for Error Trace Compaction

Due to the limited capability of the SAT-solvers, our optimum algorithm can suffer from difficulty when handling lengthy error traces in complex designs. In this section, we present two robust heuristics to overcome the shortage of the optimum algorithm. Although the heuristics cannot guarantee to obtain the shortest lengths for the error traces, experimental results demonstrate that they can still acquire incredible results within acceptable runtime.

5.1 Bounded Compaction

The first heuristic is bounded compaction. That is, we confine the maximum k value for the WALK function in binary search algorithm. Fig. 3 shows the pseudocode of the approach.

In Fig. 3, the algorithm inputs the vector of the state sequence for the original error trace, named *ET_Vec*, where the first element *ET_Vec[0]* represents the initial state and the last element *ET_Vec[n]* represents the error state.

```
// ET_Vec: the state vector of the original error trace
// BounD: the upper bound for the WALK function
// n: the original length of the error trace
// n': the compact length from ET_Vec[0] to ET_Vec[n]
01  BOUND-COMPACTION-BIN ( ET_Vec , BounD , n ) {
02      Src = ET_Vec[0] ;
03      Index = BounD ;
04      do {
05          Dest = ET_Vec[Index] ;
06          L' = OETC-BIN-SEARCH ( BounD , Src , Dest ) ;
07          if ( Index == n )
08              break ;
09          else if ( Index + (BounD - L') ≤ n )
10              Index = Index + (BounD - L') ;
11          else
12              Index = n ;
13      } while ( L' != BounD ) ;
14      n' = L' + (n - Index) ;
15      PRINT-PATH ( n' , ET_Vec[0] , ET_Vec[n] ) ;
16      return n' ;
17  }
```

Fig. 3. The bounded compaction approach.

Moreover, the algorithm inputs the length n of the original error trace and a specific upper bound, named *BounD*. Initially, the approach sets *ET_Vec[0]* as the source state *Src*. Then, it sets the *Index* value for *ET_Vec* to identify the destination state $Dest = ET_Vec[Index]$. Afterward, the heuristic calls the OETC-BIN-SEARCH procedure at line 6, which is just the optimum algorithm shown in Fig. 2. Since the maximum input length for the optimum procedure is limited to *BounD*, the approach ensures efficiency for runtime.

To clearly explain the loop from line 4 to line 13 in Fig. 3, we use Fig. 4 to show the concept. In the beginning, Fig. 4a presents the first *Index* for the destination state, S^i. The distance between the initial state and S^i is just *BounD*. Then, the heuristic performs OETC-BIN-SEARCH to obtain the compact length L' from the source state *Src* to the destination state *Dest*, which is shown in Fig. 4b. Note that L' is the shortest distance between *Src* and *Dest*. Now, the value of L' and the location of the *Index* determine three different cases:

1. If L' is smaller than *BounD* and the location of *Index* is not the error state, then the heuristic will update the *Index* value by the equation $Index = Index + (BounD - L')$, as shown at line 10 in Fig. 3. Next, the approach attempts to identify another destination state, S^j, to operate the OETC-BIN-SEARCH procedure again. Fig. 4c shows the case where the distance between S^i and S^j must be $(BounD - L')$.
2. If L' is equal to *BounD*, then the loop shown in Fig. 3 will terminate and the reduced length from the initial state to the error state will be L' plus $(n - Index)$. Fig. 4d illustrates this case.
3. If *Index* just indicates the error state, as shown at line 7 in Fig. 3, then L' will be the optimum solution. That is, L' is the shortest length from the initial state to the error state. Fig. 4e depicts the case.

In short, Fig. 4d and 4e present two possible results after applying the heuristic.

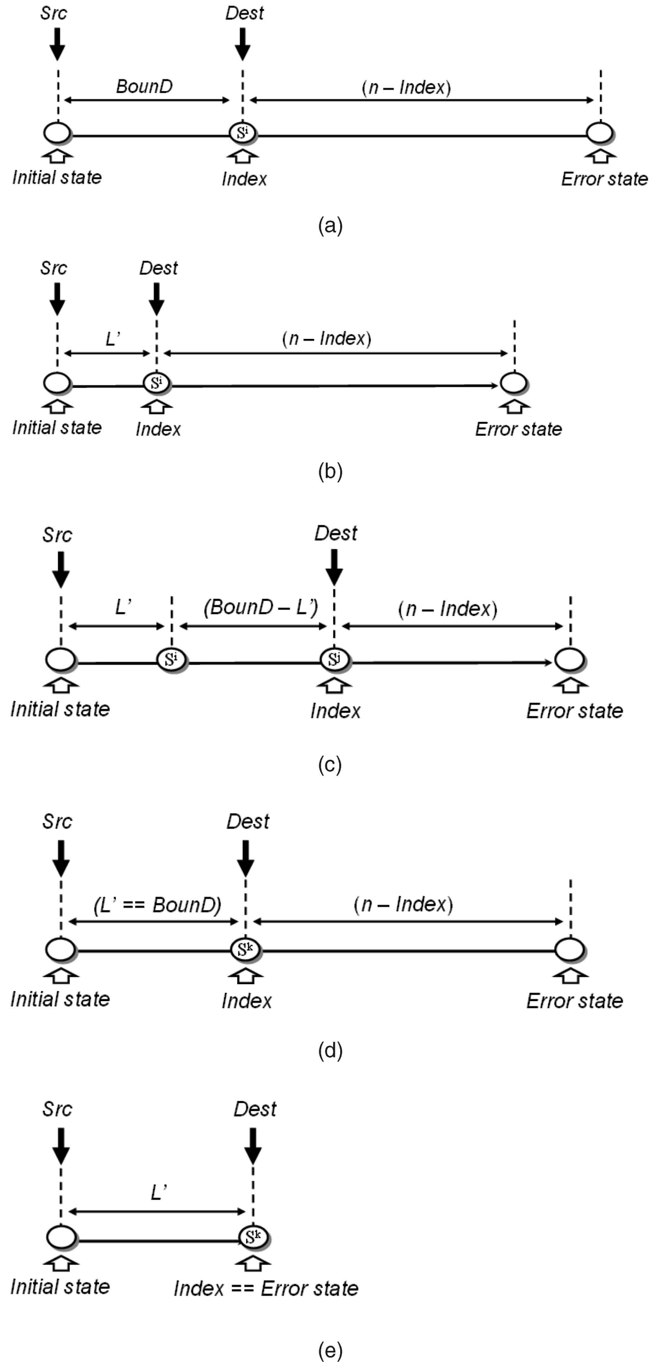


Fig. 4. The notion of the bounded compaction approach.

We take an example shown in Fig. 5 to go through the approach. In Fig. 5, the original length of the error trace is 24; the initial state and the error state are S^0 and S^{24}, respectively. If we let the *BounD* value be 10, the bounded compaction heuristic will undergo the following steps:

Step 1: $Src = S^0$, $Index = 10$, $Dest = S^{10}$. We obtain $L' = 6$.

Step 2:

$$Index = Index + (BounD - L') = 10 + (10 - 6) = 14.$$

Step 3: $Src = S^0$, $Index = 14$, $Dest = S^{14}$. We obtain $L' = 2$.

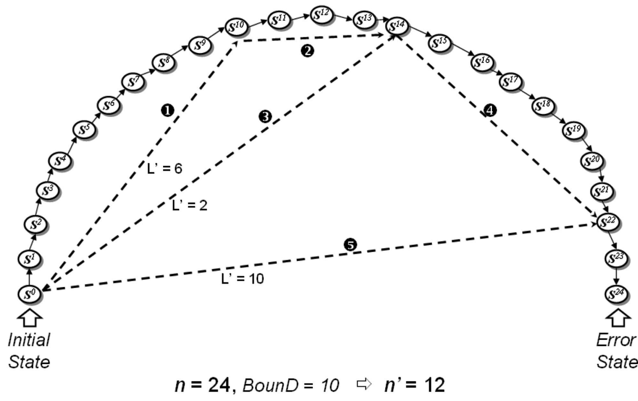


Fig. 5. An example of operating the bounded compaction approach.

Step 4:

$$Index = Index + (BounD - L') = 14 + (10 - 2) = 22.$$

Step 5: $Src = S^0$, $Index = 22$, $Dest = S^{22}$. We obtain $L' = 10$.

Step 6: Since L' is equal to *BounD*, we obtain

$$n' = L' + (n - Index) = 10 + (24 - 22) = 12.$$

Finally, we compact the error trace shown in Fig. 5 from the original length 24 to the reduced length 12.

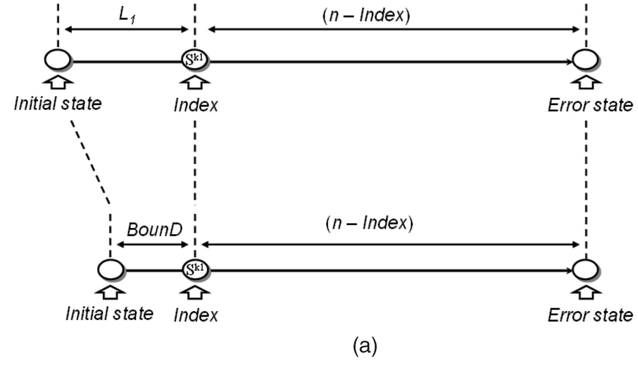
5.2 Dynamic Divide and Conquer

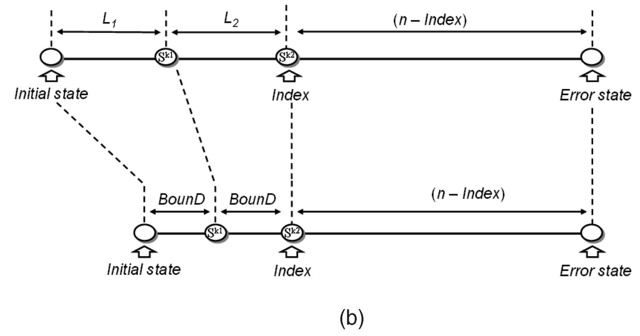
The second heuristic is the dynamic divide-and-conquer approach. Intuitively, divide and conquer is the natural way to cope with the problem of error trace compaction. That is, we can partition the original error trace into several parts and then each part can be handled separately and efficiently.

Fig. 6 shows the concept of the divide-and-conquer heuristic. For each divided segment, we apply the bounded compaction heuristic (see Fig. 3 and Fig. 4) to gain the reduced length. Initially, as indicated in Fig. 6a, we use the bounded compaction approach to reduce the segment L_1 to the length *BounD* from the initial state to the destination state S^{k1}. As described in Fig. 4d, the length L_1 is determined by the location of the *Index* value. In other words, the length of each divided segment is dynamically determined.

Next, we intend to compact the rest portion, i.e., the segment between the state S^{k1} and the error state. We set the state S^{k1} as the first element of *ET_Vec* and the error state as the last element of *ET_Vec*. In other words, we attempt to apply the bounded compaction heuristic to cope with the error trace starting at S^{k1} and ending at the error state. Fig. 6b depicts the result of this compaction, where the segment L_2 is reduced to the length *BounD* from S^{k1} to the destination state S^{k2}.

Accordingly, if we recursively handle the rest portion, we can finally obtain the compaction shown in Fig. 6c. Moreover, as Fig. 4e illustrates, the length of the last segment must be reduced to L'. If there are k segments generated in the dynamic divide-and-conquer heuristic, then the length of the compact error trace will be $(BounD * (k - 1)) + L'$.





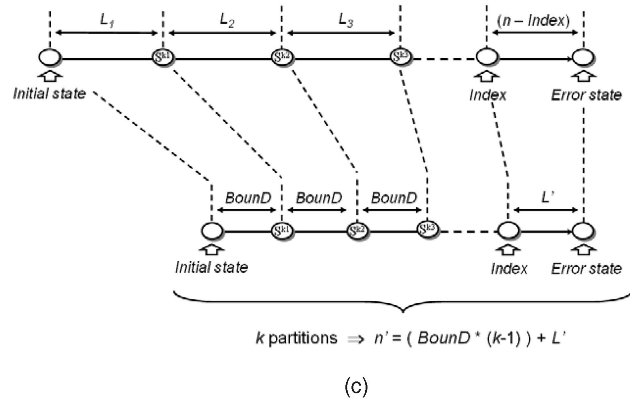


Fig. 6. The dynamic divide-and-conquer approach.

All in all, unlike the typical divide-and-conquer approach which partitions the solution space statically, our approach divides the segments of the original error trace dynamically. Each segment is handled one after another. Thus, the dynamic divide-and-conquer heuristic amplifies the power of the bounded compaction approach.

Although elegant, the dynamic divide-and-conquer heuristic still has an issue when applying the bounded compaction approach. Specifically, if the first element of *ET_Vec* is not a self-transition state, then Theorem 1 and Theorem 2 will not hold and, hence, the OETC-BIN-SEARCH procedure shown in Fig. 2 and Fig. 3 cannot operate. In such a situation, we need to use the sequential search-based algorithm for the bounded compaction heuristic. Fig. 7 and Fig. 8 depict the pseudocode of the approach.

In Fig. 7, all the codes are the same as those shown in Fig. 3, except for two parts: line 1 (BOUND-COMPACTION-SEQ) and line 6 (OETC-SEQ-SEARCH). At line 6, the procedure

```
// ET_Vec: the state vector of the original error trace
// BounD: the upper bound for the WALK function
// n: the original length of the error trace
// n′: the compact length from ET_Vec[0] to ET_Vec[n]
01  BOUND-COMPACTION-SEQ ( ET_Vec , BounD , n ) {
02      Src = ET_Vec[0] ;
03      Index = BounD ;
04      do {
05          Dest = ET_Vec[Index] ;
06          L′ = OETC-SEQ-SEARCH ( BounD , Src , Dest ) ;
07          if ( Index == n )
08              break ;
09          else if ( Index + (BounD - L′) ≤ n )
10              Index = Index + (BounD - L′) ;
11          else
12              Index = n ;
13      } while ( L′ != BounD ) ;
14      n′ = L′ + (n - Index) ;
15      PRINT-PATH ( n′ , ET_Vec[0] , ET_Vec[n] ) ;
16      return n′ ;
17  }
```

Fig. 7. The bounded compaction approach by applying the sequential search manner.

OETC-SEQ-SEARCH operates the sequential search-based algorithm for compacting error traces. Fig. 8 illustrates such procedure. By employing the sequential search-based bounded compaction heuristic, we can perform the dynamic divide-and-conquer approach whether *ET_Vec[0]* is a self-transition state or not.

In the end, we apply Fig. 9 to summarize the dynamic divide-and-conquer approach. In Fig. 9, segments L_1 and L_3 can be handled by the binary search-based method, BOUND-COMPACTION-BIN, since the initial state and the intermediate state S^{k2} are self-transition states. On the other hand, segments L_2 and L_4 must be handled by the sequential search-based method, BOUND-COMPACTION-SEQ, since neither the state S^{k1} nor S^{k3} is a self-transition state.

5.3 Experimental Results

We implemented the two heuristics in C++. To demonstrate their efficiency, we first conducted the largest error trace in Table 2—s5378-3. Table 3 summarizes the results, where the first six columns record the data of *CET2* and the optimum algorithm, named *OETC*. The seventh column, entitled "BounD," stands for the parameter used in the two heuristics. In this experiment, the *BounD* value ranged from 10 to 50.

The eighth, ninth, and tenth columns present the reduced lengths, the runtime, and the reduction ratio by

```
// IS: the source state
// ES: the destination state
// n: the original length from IS to ES
// n′: the compact length from IS to ES
01  OETC-SEQ-SEARCH ( n , IS , ES ) {
02      n′ = 1 ;
03      while ( n′ <  n ) {
04          if ( PATH (n′, IS, ES) == true )
05              break ;
06          n++ ;
07      }
08      PRINT-PATH ( n′ , IS , ES ) ;
09      return n′ ;
10  }
```

Fig. 8. The sequential search-based optimum error trace compaction algorithm.

applying the bounded compaction heuristic. For $BounD = 10$, we see that, although the approach took very little runtime (17.71 seconds), it acquired a very low reduction ratio. However, as the *BounD* value increases, the reduction ratio augments accordingly. When $BounD = 50$, the approach even obtained the optimum solution, while it just took about 300 seconds (316.10 seconds).

Similarly, Table 3 also presents the reduced length, the runtime, and the reduction ratio by applying the dynamic divide-and-conquer heuristic. Since the approach iteratively employs bounded compaction, the results were superior to those obtained by bounded compaction, although it consumed more runtime. However, the time used in the approach was still much less than that of *CET2* and *OETC*.

Furthermore, when $BounD = 50$, the dynamic divide-and-conquer method operated with the same runtime as the bounded compaction approach. This is because the first divided segment in the dynamic divide-and-conquer method can handle the state sequence from the initial state to the error state and, hence, no additional part can be handled further.

Next, we conducted a real design to evaluate the effectiveness of our heuristics. The design, named SCPU, is a subset of a microprocessor. It contains 1,638 sequential elements and 37,464 gates. In this experiment, we generated 10 error traces for compaction. Moreover, the maximum runtime was limited to 100,000 seconds and the *BounD* value used in the two heuristics was 30. Table 4 shows the comparison of the results of the four methods (*CET2*, *OETC*, Bounded Compaction, and Dynamic Divide and Conquer). The empty cells indicate that the results were not available within the runtime limit.

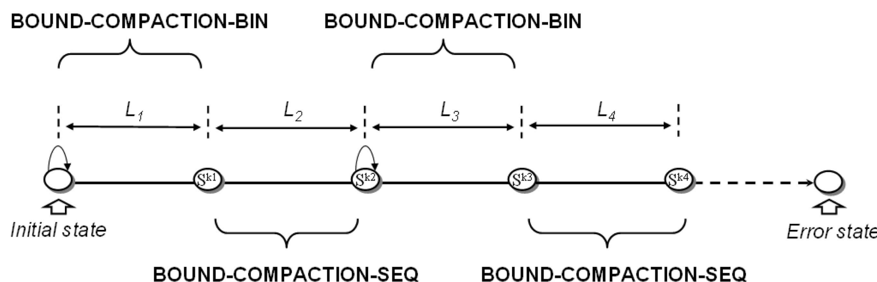


Fig. 9. Handling divided segments the in dynamic divide-and-conquer approach.

TABLE 3
Experimental Results of Applying Heuristics to s5378-3

Error Trace: s5378-3 (n = 607)

CET2			OETC			BounD	Bounded Compaction			Divide and Conquer		
Length	Time	Ratio	Length	Time	Ratio		Length	Time	Ratio	Length	Time	Ratio
120	5763.71	80.23%	36	95951.30	94.07%	10	597	17.71	1.65%	165	201.01	72.82%
						20	119	127.06	80.39%	91	266.79	85.01%
						30	52	191.51	91.43%	36	193.00	94.07%
						40	52	245.46	91.43%	52	255.10	91.43%
						50	36	316.10	94.07%	36	316.10	94.07%

TABLE 4
Comparison of the Four Error Trace Compaction Methods

TIME_LIMIT = 100000 s BounD = 30

	Error Trace		CET2		OETC		Bounded Compaction		Divide and Conquer	
	Name	Length	Length	Ratio	Length	Ratio	Length	Ratio	Length	Ratio
SCPU	S-001	226	226	0%	24	89.38%	24	89.38%	24	89.38%
	S-002	404	404	0%	-	-	225	44.31%	47	88.37%
	S-003	538	538	0%	-	-	398	26.02%	50	90.71%
	S-004	699	699	0%	-	-	242	65.38%	47	93.28%
	S-005	864	-	-	-	-	364	57.87%	102	88.19%
	S-006	1005	-	-	-	-	916	8.86%	246	75.52%
	S-007	1396	-	-	-	-	892	36.10%	715	48.78%
	S-008	1812	-	-	-	-	1604	11.48%	1197	33.94%
	S-009	2235	-	-	-	-	2066	7.56%	1177	47.34%
	S-010	3588	-	-	-	-	3520	1.90%	1803	49.75%

Obviously, the *CET2* and *OETC* approaches cannot finish the 10 tasks of SCPU. *OETC* only finished the smallest task even though the result is guaranteed to be the shortest length. For *CET2*, although it handled more tasks, it contributed no improvement to error trace compaction since all the finished tasks acquired 0 percent reduction ratio.

Unlike *CET2* and *OETC*, the bounded compaction heuristic, as well as the dynamic divide-and-conquer approach, completed all 10 error traces of SCPU. In fact, the bounded compaction method even finished the 10 tasks within 30,000 seconds. However, the performance of the bounded compaction method is not very desirable. For the error trace S-006, S-009, and S-010, the bounded compaction heuristic just obtained no more than 10 percent reduction ratio.

On the other hand, the dynamic divide-and-conquer approach acquired an incredible reduction ratio for each compaction. The major reason, as described in Fig. 6, is because the dynamic divide-and-conquer method can further compact the portion that the bounded compaction approach cannot handle.

6 Conclusions and Future Work

We have presented an optimum algorithm that takes the paradigm of binary search for compacting error traces. The algorithm not only ensures $\log_2(n)$ number of operations, but also guarantees to gain the shortest length from the initial state to the error state. Based on the algorithm, we also present two robust and practical heuristics to handle real designs. Although not optimum, experimental results truly show that our approaches outperform prior work. We believe that, through the manipulation of our approaches, designers can earn numerous profits to increase their debugging performance. Future work will concentrate on developing more powerful heuristics to speed up the compaction. Moreover, applying other symbolic techniques to enhance the efficiency can be another practical issue.

REFERENCES

[1] H. Foster, A. Krolnik, and D. Lacey, *Assertion-Based Design.* Kluwer Academic, 2003.

[2] B. Cohen, S. Venkataramanan, and A. Kumari, *Using PSL/Sugar with HDL for Formal and Dynamic Verification.* VhdlCohen Publishing, 2004.

[3] J.R. Burch, E.M. Clarke, D.E. Long, K.L. McMillan, and D.L. Dill, "Symbolic Model Checking for Sequential Circuit Verification," *IEEE Trans. Computer-Aided Design of Integrated Circuits,* vol. 13, no. 4, pp. 401-424, Apr. 1994.

[4] A. Biere, A. Cimatti, E.M. Clarke, M. Fujita, and Y. Zhu, "Symbolic Model Checking Using SAT Procedures instead of BDDs," *Proc. 36th Design Automation Conf.,* pp. 317-320, 1999.

[5] M.S. Hsiao, E.M. Rundnick, and J.H. Patel, "Fast Static Compaction Algorithms for Sequential Circuit Test Vectors," *IEEE Trans. Computers,* vol. 48, no. 3, pp. 311-322, Mar. 1999.

[6] E.M. Rundnick and J.H. Patel, "Efficient Techniques for Dynamic Test Sequence Compaction," *IEEE Trans. Computers,* vol. 48, no. 3, pp. 323-330, Mar. 1999.

[7] Y. Chen and F. Chen, "Algorithms for Compacting Error Traces," *Proc. Eighth Asia and South Pacific Design Automation Conf.,* pp. 99-103, 2003.

[8] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, *Introduction to Algorithms.* MIT Press, 2003.

[9] M. Mneimneh and K. Sakallah, "Computing Vertex Eccentricity in Exponentially Large Graphs: QBF Formulation and Solution," *Proc. Sixth Int'l Conf. Theory and Applications of Satisfiability Testing,* 2003.

[10] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an Efficient SAT Solver," *Proc. 38th Design Automation Conf.,* pp. 530-535, 2001.

[11] M. Mneimneh and K. Sakallah, "SAT-based Sequential Depth Computation," *Proc. Eighth Asia South Pacific Design Automation Conf.,* pp. 87-92, 2003.

[12] M.K. Ganai, A. Aziz, and A. Kuehlmann, "Enhancing Simulation with BDDs and ATPG," *Proc. 36th Design Automation Conf.,* pp. 385-390, 1999.

Chia-Chih Yen received the BS degree in electrical engineering from National Taiwan University and the MS degree in electronics engineering from National Chiao Tung University. He is a PhD candidate in the Department of Electronics Engineering at the National Chiao Tung University, Hsinchu, Taiwan. His research interests include formal and semiformal design verification.

Jing-Yang Jou received the BS degree in electrical engineering from National Taiwan University, Taiwan, Republic of China, and the MS and PhD degrees in computer science from the University of Illinois at Urbana-Champaign in 1979, 1983, and 1985, respectively. He is currently the Director General of the National Chip Implementation Center, National Applied Research Laboratories in Taiwan. He is a full professor and was chairman of the Electronics Engineering Department from 2000 to 2003 at National Chiao Tung University, Hsinchu, Taiwan. Before joining Chiao Tung University, he was with GTE Laboratories from 1995 to 1996 and with AT&T Bell Laboratories in Murray Hill, New Jersey, from 1986 to 1994. His research interests include logic and physical synthesis, design verification, CAD for low power, and network on chips. He has published more than 160 technical papers. Dr. Jou is a fellow of the IEEE. He has been elected to be president of the Taiwan Integrated Circuit Design Society (TICD) 2007-2008. He was the technical program chair of the Asia-Pacific Conference on Hardware Description Languages (APCHDL '97), the technical program chair of the 12th VLSI Design/CAD Symposium (2001), the executive chair of the Second Taiwan-Japan Microelectronics International Symposium (2002), and the honorary chair of the International Workshop on Multi-Project Chip (IWMC '06). He was the recipient of the distinguished paper award of the IEEE International Conference on Computer-Aided Design in 1990 and the Outstanding Academy-Industry Cooperation Achievement Award granted by the Ministry of Education (MOE), Taiwan, in 2002.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.