Test Compaction for Synchronous Sequential Circuits by Test Sequence Recycling

Irith Pomeranz and Sudhakar M. Reddy +
Electrical and Computer Engineering Department
University of Iowa
Iowa City, IA 52242

Abstract

We introduce a new concept for test sequence compaction referred to as *recycling*. Recycling is based on the observation that easy-to-detect faults tend to be detected several times by a deterministic test sequence, whereas hard-to-detect faults are detected once towards the end of the test sequence. Thus, the suffix of a test sequence detects a large number of faults, including hard-to-detect faults. The recycling operation keeps a suffix S_1 of a test sequence T_1 and discards the rest of the sequence. The suffix S_1 is then used as a prefix of a new test sequence T_2. In this process, S_1 is expected to detect the more difficult to detect faults as well as many of the easy-to-detect faults, resulting in a new sequence T_2 which is shorter than T_1. Recycling is enhanced by a scheme where several faults are targeted simultaneously to generate the shortest possible test sequence that detects all of them.

1. Introduction

Several dynamic and static test sequence compaction procedures for synchronous sequential circuits have been proposed recently. Dynamic compaction [1]-[4] incorporates into the test generation procedure techniques for reducing the final test length. Static compaction techniques [5]-[7] are applied as a postprocessing step to test generation, and can be applied in addition to or instead of dynamic compaction to reduce the length of the test sequences produced by a test generation procedure.

In this work, we introduce a new concept for test sequence compaction referred to as *recycling*. It can be viewed as a hybrid of dynamic and static compaction techniques. Recycling is based on the well-known observation that easy-to-detect faults tend to be detected by the test vectors produced early during the test generation process. These faults are detected several times along the test sequence, leaving difficult-to-detect faults to be detected only once towards the end of the test sequence. Although fault ordering may affect the time unit in which a fault is detected, test generation is typically applied only to a relatively small number of faults; the remaining faults are accidentally detected by the test sequence generated for the targeted faults. Thus, for most of the faults, the time unit where they are detected is a reasonable measure of the difficulty of detecting them. The prefix of a test sequence (i.e., a subsequence that starts at the beginning of the test sequence) that contributes only to the detection of easy-to-detect faults may unnecessarily lengthen the test sequence. However, it may not be possible to omit a prefix of the test sequence without losing fault coverage. To take advantage of these observations, recycling keeps a *suffix* S_1 of a test sequence T_1 (i.e., it keeps a subsequence S_1 that ends at the end of the sequence T_1) and discards the rest of the sequence. The suffix is selected such that it detects the most difficult to detect faults. For the remaining faults, test generation starts from the sequence S_1, and S_1 is extended into a complete test sequence T_2 with S_1 as a *prefix* of T_2 (hence, this process *recycles* the most useful part of the test sequence into a new test sequence). In this process, S_1 is expected to detect the more difficult to detect faults as well as many of the easy-to-detect faults, resulting in a sequence T_2 which is shorter than T_1. The recycling process may now be repeated using a suffix S_2 of T_2. The suffix S_2 of T_2 is selected such that the sequence S_1S_2 detects the most difficult to detect faults. Test generation starting from S_1S_2 is used to obtain a test sequence T_3 which is expected to be shorter than T_2. This may be repeated any number of times.

The main difference between recycling and static compaction is that static compaction procedures manipulate a given test sequence and do not perform additional test generation, whereas recycling repeats the test generation process for faults that remain undetected by the selected suffixes. Thus, recycling has more flexibility in reducing the test length. Compared to other dynamic compaction techniques, recycling requires only minor modifications to the underlying test generation process, as described in Section 2. Thus, it is easier to implement. In addition, it is general purpose in the sense that any test generation procedure can be plugged into the recycling process to further reduce the test sequences it generates.

Recycling requires repeated test generation. It is possible to keep this test generation effort to a minimum by using the following observations. (1) Only faults that are not detected by the suffix selected for inclusion in the new test sequence need to be considered again. By keeping the number of faults detected by the suffix large, the additional test generation effort can be minimized. (2) Faults that could not be detected in the first test generation pass (undetectable and aborted faults) need not be considered again for test generation. In cases where the test generation process is not complete, undetected faults may be simulated, to take advantage of accidental detection.

Recycling can be applied in conjunction with any test generation procedure. In this work, we apply recycling to the test sequences generated by the basic test generation procedure described in [1]. This basic procedure does not include any of the compaction heuristics introduced in [1]. Nevertheless, the lengths of the test sequences it generates are typically shorter than those of other test generation procedures that do not employ compaction techniques, such as [8]. Thus, it provides a good basis for experimenting with the recycling procedure. We also

+ Research supported in part by NSF Grant No. MIP-9357581 and NSF Grant No. CDA-9601503.

0-8186-8409-7/98 $10.00 © 1998 IEEE

modify the basic procedure of [1] to consider several faults simultaneously, thus generating shorter test sequences. Recycling is shown to be effective both before and after this modification of the basic procedure. Recycling can be applied together with static and/or dynamic test compaction procedures to obtain even shorter test sequences.

The paper is organized as follows. In Section 2 we describe the general recycling concept and a specific recycling procedure. In Section 3 we describe a complete test generation procedure based on recycling. Section 4 includes experimental results. Section 5 concludes the paper.

2. Test sequence recycling

The goal of test sequence recycling is to repeatedly extract from a test sequence the parts that detect the most difficult to detect faults, and use them in constructing a new and shorter test sequence. In this section, we first describe a generic recycling procedure. In this procedure, we use hard-to-detect faults and subsequences to detect them, without defining how these faults are identified or how the subsequences are derived. We then describe a specific implementation that incorporates definitions of hard-to-detect faults and subsequences to detect them.

The generic recycling procedure is given below as Procedure 1. Initially, we have a set of faults F, and empty sequences T and S. At every iteration of the procedure, T is set equal to S and then extended by test generation into a test sequence for all the faults in F. In the first iteration, $S = \phi$ and test generation is carried out for all the faults in F. In the following iterations, $S \neq \phi$, and test generation is carried out only for faults not detected by S. Once T is complete, a subset of hard-to-detect faults F_0 is selected. A subsequence S' is extracted from T such that $S \cdot S'$ detects all the faults in F_0 (here, $\cdot$ stands for concatenation). The sequence S is replaced by the sequence $S \cdot S'$, and the new sequence S is used in the next iteration. This process is repeated until the length of $S \cdot S'$ becomes equal to or larger than the length of T, or $S \cdot S'$ detects all the faults in F. Throughout the procedure, whenever a complete test sequence T is generated, its length is compared to the length of the shortest test sequence generated so far, stored in T_{best}. If T is shorter than T_{best}, then T is stored in T_{best}. This step is needed since the test sequence length is not guaranteed to decrease monotonically with additional iterations of the procedure.

Procedure 1: A generic recycling process

(1) Let F be the set of target faults. Let $T = (\phi)$ (an empty sequence). Let $S = (\phi)$. Let $T_{best} = (\phi)$.

(2) Extend T to detect the faults in F. If $T_{best} = (\phi)$ or $|T| < |T_{best}|$, store T in T_{best} ($|A|$ is the length of the sequence A).

(3) Select a set of hard-to-detect faults $F_0 \subseteq F$.

(4) Extract from T a subsequence S' such that $S \cdot S'$ detects all the faults in F_0.

(5) If $|S \cdot S'| \geq |T|$, stop: T_{best} is the final test sequence.

(6) Set $S = S \cdot S'$ and $T = S$.

(7) Simulate F under T and drop the detected faults from F.

(8) If $|F| = \phi$:

  (a) If $|T| < |T_{best}|$, store T in T_{best}.

  (b) Stop: T_{best} is the final test sequence.

(9) Go to Step 2.

Note that since S is always kept as a prefix of T, the faults it detects can be dropped from further consideration. This is done in Step 7 of Procedure 1.

Next, we describe a specific implementation of Procedure 1. In this implementation, the subset of hard-to-detect faults F_0 consists of a single fault f_0. A similar procedure can be used for $|F_0| > 1$. We use the detection time of a fault (the time unit at which the fault is detected for the first time) as a measure of the difficulty in detecting it. Thus, the most difficult to detect fault f_0 is the one with the highest detection time. Other definitions of hard-to-detect faults can be used. We extract the shortest suffix of the test sequence that detects f_0 by performing binary search over all possible suffixes. A suffix S_1' is said to detect f_0 if S_1' detects f_0 when the fault free circuit and the faulty circuit in the presence of f_0 are started from their all-unspecified initial states before S_1' is applied. Binary search will find the shortest possible suffix due to the following property, which results from the fact that the all-unspecified initial states are used for the suffixes.

Lemma 1: If S_1' and S_1'' are two suffixes of a test sequence T such that S_1' is shorter than S_1'', then S_1'' detects every fault detected by S_1'.

To show that binary search will find the shortest possible suffix, let S_1 be the shortest suffix of T that detects f_0. Consider an arbitrary step where a suffix S_1' is considered. If S_1' is longer than S_1, then S_1' detects f_0 and binary search will continue to consider a suffix shorter than S_1'; and if S_1' is shorter than S_1, then S_1' does not detect f_0 and binary search will continue to consider a longer suffix. The suffix length will continue to change until binary search converges to S_1.

To describe the recycling process in more detail, we use the following notation. The input vector at time unit u of a sequence A is denoted by $A[u]$. The subsequence of A from time unit u_1 to time unit u_2 is denoted by $A[u_1 \cdots u_2]$. The first time unit where a fault f is detected by a test sequence T is denoted by $u_{det}(f)$. The length of T, $|T|$, is denoted by L. Binary search to find the shortest suffix that detects a fault f_0 is done using three variables. Variable *low* holds the index of the first time unit where the suffix may start; variable *high* holds the index of the last time unit where the suffix may start; and variable *mid* is computed as $mid = (low + high)/2$. The following example illustrates the proposed implementation of the recycling process.

We consider ISCAS-89 benchmark circuit $s27$. The test sequence T_1 shown in Table 1(a) is first generated to detect all 32 faults in the circuit. Next to every input vector $T_1[u]$ we show in Table 1(a) the indices of the faults detected at the same time unit u (faults with $u_{det}(f) = u$). The fault with the highest detection time is fault 25, with $u_{det}(f_{25}) = 16$. We therefore concentrate on this fault. Our goal is to find the shortest suffix S_1 of T_1 that detects f_{25}. To find the suffix, we use binary search over T, and we simulate f_{25} under the following sequences.

Initially, $low = 0$, $high = 16$ and $mid = (0 + 16)/2 = 8$. We obtain the suffix $T_1[8 \cdots 16]$. The fault f_{25} is detected by this sequence, therefore, we set $low = mid + 1 = 9$. We now have $mid = (9 + 16)/2 = 12$ and we consider the suffix $T[12 \cdots 16]$. The fault f_{25} is detected by this sequence as well, therefore, we set $low = 13$, $mid = 14$, and we consider the suffix $T[14 \cdots 16] = (0010, 1001, 0000)$. This sequence still detects f_{25}. We therefore obtain $low = 15$ and $mid = 15$, and we consider the suffix $T[15 \cdots 16] = (1001, 0000)$. This sequence does not detect f_{25}. Therefore, we update $high = 14$. We now have $high < low$, and binary search terminates. The shortest suffix to detect f_{25} is $S_1 = T[high \cdots L-1]$, or $T[14 \cdots 16] = (0010, 1001, 0000)$.

Table 1: An example of recycling for $s27$

(a) The test sequence T_1

u	$T[u]$	detected
0	0010	
1	1101	0,9,15,20,22,24,30
2	0010	
3	0001	1,2,6,8,11,12,17,23,29,31
4	1000	7,13,21
5	0001	3,28
6	0000	4,27
7	0100	
8	0001	5,16
9	0010	
10	0000	
11	0001	10
12	0000	14,18,26
13	0100	19
14	0010	
15	1001	
16	0000	25

(b) The test sequence T_2

u	$T[u]$	detected
0	0010	
1	1001	2,6,8,11,17,23,29,31
2	0000	14,18,25,26
3	1101	0,7,9,13,15,20,22,24,30
4	0010	
5	0001	1,12
6	1000	21
7	0001	3,28
8	0000	4,27
9	0100	
10	0001	5,16
11	0010	
12	0000	
13	0001	10
14	0100	19

Next, we simulate S_1 and find that it detects 12 faults. For the remaining 20 faults, we extend S_1 into a test sequence T_2 by performing test generation. The resulting test sequence is shown in Table 1(b). Note that S_1 is now a prefix of the test sequence. The new test sequence T_2 is of length 15, shorter by two test vectors than the original sequence.

We now repeat the recycling process with the fault f_{19} detected by T_2 at time unit 14. Our goal is to select a suffix S_2 such that $S_1 \cdot S_2$ detects f_{19}. Binary search starts with $low = 3$ (since $T[0 \cdots 2] = S_1$ must be left intact), $high = 14$ and $mid = (3 + 14)/2 = 8$, and we consider the following sequences.

We first consider $S_2 = T_2[8 \cdots 14]$. We simulate f_{19} under the sequence $S_1 \cdot S_2$ = (0010, 1001, 0000, 0000, 0100, 0001, 0010, 0000, 0001, 0100). Since f_{19} is detected by this sequence, we set $low = 9$, $mid = 11$, and we continue to consider the suffix $S_2 = T_2[11 \cdots 14]$. Continuing this process, we find that f_{19} is detected using the suffix $S_2 = [14]$. This is the suffix selected.

Next, we simulate $S_1 \cdot S_2$ = (0010, 1001, 0000,0100) and find that it detects 13 faults. The new test sequence T_3 is constructed for the remaining 19 faults starting from $S_1 \cdot S_2$. The resulting test sequence is of length 11.

The next recycling step does not improve the test length any further. In this case, two iterations of recycling reduce the test length from 17 to 11.

In the previous example, the test sequence length reduced monotonically with every additional recycling step. In general, this may not be the case. Consequently, one must save the test sequence every time a test sequence is obtained which is shorter than any previously computed test sequence. The recycling procedure is summarized next as Procedure 2.

Procedure 2: The recycling process

(1) Let F be the set of target faults. Let $T = (\phi)$. Let $L_{pref} = 0$ (L_{pref} is the total length of all the suffixes concatenated to form the prefix of the new test sequence).

(2) Extend T to detect the faults in F. For every fault f, maintain its first detection time in $u_{det}(f)$. If f is not detected, set $u_{det}(f) = -1$. Let the length of T be L. Let $L_{old} = L$. If $T_{best} = (\phi)$, or T is shorter than T_{best}, or T detects more faults than T_{best}, store T in T_{best}.

(3) Select a fault f_0 such that $u_{det}(f_0) = \max\{u_{det}(f): f \in F\}$.

(4) Set $low = L_{pref}$ and $high = L - 1$.

(5) Set $mid = (low + high)/2$.

(6) Let $T' = T[0 \cdots L_{pref} - 1] \cdot T[mid \cdots L - 1]$.

(7) If T' detects f_0, set $low = mid + 1$. Else, set $high = mid - 1$.

(8) If $high \geq low$, go to Step 5.

(9) Let $T = T[0 \cdots L_{pref} - 1] \cdot T[high \cdots L - 1]$. Let L be the length of T. Let $L_{pref} = L$.

(10) If $L = L_{old}$, stop: T_{best} is the final test sequence.

(11) Simulate F under T and drop the detected faults from F.

(12) If $|F| = \phi$:

  (a) If $|T| < |T_{best}|$, or T detects more faults than T_{best}, store T in T_{best}.

  (b) Stop: T_{best} is the final test sequence.

(13) Go to Step 2.

Note that since the prefix $T[0 \cdots L_{pref} - 1]$ is always kept intact as L_{pref} is increased, there is no need to resimulate the faults detected by the prefix. These faults are dropped from the set of target faults F in Step 11. In addition, if a fault f cannot be detected in Step 2 of Procedure 2, it can be marked and test generation for it does not need to be done again.

3. The complete test generation procedure

In this section, we complete the description of our implementation of Procedure 2 by describing the test generation procedure used in Step 2 of Procedure 2. The first test generation procedure we use is the basic procedure used in [1] (before adding dynamic compaction heuristics to it). We also modify this procedure to generate shorter test sequences by considering several faults simultaneously.

In the basic test generation procedure of [1], test generation starts from an initial test sequence T (T may be empty). At every iteration of the procedure, a target fault f is selected. The test sequence T is simulated in the fault free circuit and in the faulty circuit in the presence of f starting from the all-unspecified initial states. If f is not detected by T, then at the end of the simulation process, the final states reached by the fault free and the faulty circuits are recorded. Let the final state of the fault free circuit be Q_0 and let the final state of the faulty circuit in the presence of f be Q'_0 (when $T = \phi$, Q_0 and Q'_0 are the all-unspecified states). A test sequence is generated for f starting from the state pair Q_0/Q'_0. If a test sequence T' is generated, T' is added at the end of T. The resulting test sequence detects f and possibly additional faults that were not detected by T. The new sequence is simulated, and all the yet-undetected faults that it detects are dropped from the list of target faults. This process is repeated for other target faults, until all the faults have been considered. This process is similar in structure to the process used by other test generation procedures such as [8] and [9].

Next, we describe the generation of a test sequence T' starting from a given state pair Q_0/Q'_0. It is convenient to describe the procedure by a tree with vertices corresponding to pairs of fault-free/faulty states and edges corresponding to state transitions. The edges are labeled by the input and output values corresponding to the state transitions. An example of such a tree is shown in Figure 1. The state pair Q_0/Q'_0 is placed in the root of the tree. The test generation procedure starts from Q_0/Q'_0 and creates the tree in BFS order, level by level. Each state pair appears at most once in the tree. If a state transition leads to a state pair that already exists in the tree, it is not included in the

tree. To limit the run time of the procedure, a limited number of randomly selected input vectors N_{vec} is applied for every state pair. In addition, the number of state pairs in each level of the tree is limited to N_{st}. The procedure terminates with the first state transition where the output vector of the fault free circuit β conflicts with the output vector of the faulty circuit β'. In Figure 1, we replace β/β' by β if $\beta = \beta'$. The edge labeled $\alpha_{31}, \beta_{31}/\beta'_{31}$ indicates that the fault can be detected by the input sequence $\alpha_{11}\alpha_{21}\alpha_{31}$. No additional state transitions are explored following the state transition that detects the fault. Alternatively, the procedure terminates when all the successors of all the state pairs reached are abandoned, or when a limit on test length is reached. In this case, the fault is aborted. The test generation procedure above is referred to as Procedure 3. When Procedure 3 is called with $T = \phi$, we refer to the procedure as the *underlying test generation procedure.*

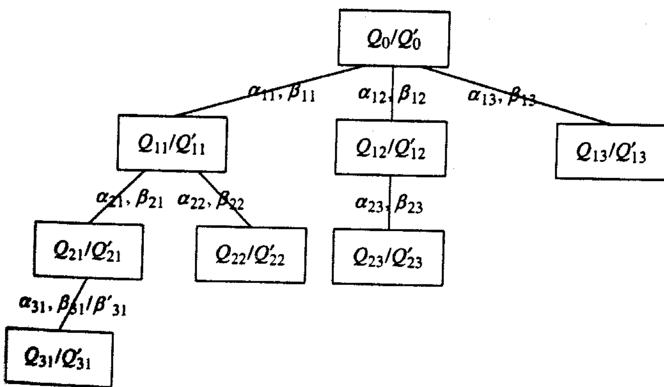

Figure 1: An example of test generation for a single fault

Next, we consider an extension of Procedure 3, aimed at generating shorter test sequences. The extended procedure considers several faults simultaneously and generates the test sequence T' to detect as many of these faults as possible. Consideration of several faults was used in [10] to generate compact test sets for combinational circuits. The following modifications are made to Procedure 3. Instead of selecting a single yet-undetected fault, N_f yet-undetected faults are selected for test generation. The test generation process for N_f faults is similar to the test generation process for a single fault. The test generation process for two faults, f_1 and f_2, is illustrated by the tree shown in Figure 2 (f_1 is the same fault considered in Figure 1). The vertices of the tree contain triples of the form $Q/Q'/Q''$, where Q is the state of the fault free circuit, Q' is the state of the faulty circuit in the presence of f_1, and Q'' is the state of the faulty circuit in the presence of f_2. The search for a test sequence terminates in one of two cases, as explained next.

The first case occurs when both faults are detected. This is demonstrated by the tree of Figure 2. The transition into state $Q_{32}/Q'_{32}/Q''_{32}$ has output vectors $\beta_{32}/\beta'_{32}/\beta''_{32}$ such that β_{32} conflicts with both β'_{32} and β''_{32}. Thus, both faults are detected by the input sequence $\alpha_{11}\alpha_{22}\alpha_{32}$. Note that based on the tree of Figure 1, we selected the test sequence $\alpha_{11}\alpha_{21}\alpha_{31}$ for f_1. Using the same sequence length, the sequence $\alpha_{11}\alpha_{22}\alpha_{32}$ detects both f_1 and f_2.

The second case where the test generation procedure for f_1 and f_2 terminates occurs when all the successors reached in the tree are abandoned, or a limit on test length is reached. In this case, we look for a state transition out of the highest level in

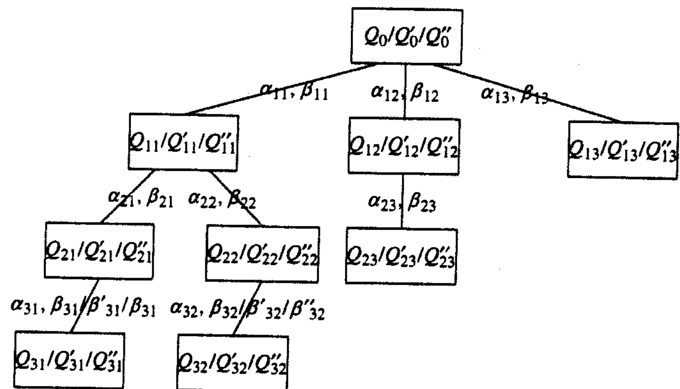

Figure 2: Example 1 of test generation for two faults

the tree (corresponding to the shortest sequence) where the largest number of faults are detected. We use this state transition to trace a test sequence T' to detect as many faults as possible.

In implementing the test generation procedure, it is necessary to keep track of cases where a fault is detected by a state transition at a level of the tree which is higher than the level currently considered. This is illustrated by Figure 3, where the first fault f_1 is detected by the state transition labeled $\alpha_{11}, \beta_{11}/\beta'_{11}/\beta_{11}$. The fault is not detected again in lower levels of the tree; however, any sequence that starts with α_{11} detects f_1. We keep track of this fact by storing in the vertices the list of faults detected by state transitions they can be reached from. Thus, the vertices $Q_{21}/Q'_{21}/Q''_{21}$ and $Q_{22}/Q'_{22}/Q''_{22}$ contain f_1. To limit the run time of the procedure, after generating a level of the tree, we omit all the vertices that detect fewer than the maximum number of faults detectable at that level. In Figure 3, only the vertex $Q_{11}/Q'_{11}/Q''_{11}$ is used to continue the test generation process.

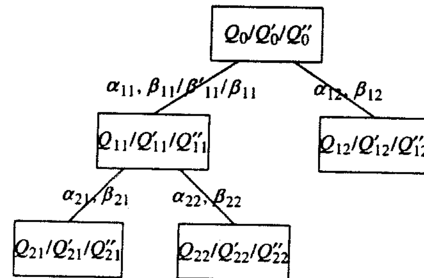

Figure 3: Example 2 of test generation for two faults

The test generation procedure above is summarized as Procedure 4. In Steps 2-6 of Procedure 4, test generation is carried out for subsets of faults of size at most N_f. During the test generation process, faults that have been considered are removed from the set F_{left}. However, faults that are not detected when they are considered as part of a subset are not declared undetected. Instead, they are considered each one alone by calling Procedure 3 in Step 8 of Procedure 4.

Procedure 4: Test generation starting from a test sequence T considering faults simultaneously

(1) Let F be the set of target faults. Set $F_{left} = F$ and $F_{det} = \phi$.
(2) Set $F_t = \phi$.
(3) While $|F_t| < N_f$ and $F_{left} \neq \phi$:

(a) Select a fault $f \in F_{left}$.

(b) Simulate f under T. Record the final states reached. If f is not detected, add f to F_t.

(c) Remove f from F_{left}.

(4) Construct a tree (such as the ones in Figures 2 and 3) for F_t starting from the final states reached in Step 3.

(5) Select the state transition where the largest number of faults are detected, and construct a test sequence T'.

(6) Add T' at the end of T.

(5) For every fault $f \in F_{left}$:
Simulate f under T. If f is detected, add f to F_{det} and remove it from F_{left}.

(6) If $F_{left} \neq \phi$, go to Step 2.

(7) Set $F_{left} = F - F_{det}$.

(8) Call Procedure 3 to extend T into a test sequence for F_{left}.

(9) Stop: T is the required test sequence.

4. Experimental results

We applied the recycling procedure, Procedure 2, to ISCAS-89 benchmark circuits. We used Procedures 3 and 4 as the test generation procedures embedded in Procedure 2. In Procedure 4, we used subsets of faults of size three. During test generation for a fault or subset of faults, we allowed the test length to be at most 100 (at most 100 levels in the test generation tree); we considered at most 100 input vectors in every state ($N_{vec} = 100$); and we allowed at most $N_{st} = 100$ state pairs in every level of the test generation tree. We allowed at most 10 recycling operations in Procedure 2 before terminating the procedure.

The first set of results is reported in Table 2 as follows. In Table 2(a) we report the results when Procedure 3 is used for test generation. After circuit name we show the number of faults in the circuit. We then show the number of faults detected and the test length for the underlying test generation procedure (before recycling). Next, we show the number of faults detected and the test length after recycling, and the number of recycling iterations required before the test length reached its minimum value. In Table 2(b) we show the same information when Procedure 4 is used as the test generation procedure and at most three faults are processed simultaneously. The following points should be noted.

Comparing the results before and after recycling in each table, it can be seen that except for *s*820, recycling is effective in reducing the test length. For *s*820, recycling increases the number of detected faults. Consequently, the test length is increased as well. The number of recycling iterations is sometimes high. The reduction of the number of iterations is considered below. It can also be seen that non-trivial reductions in test length are sometimes obtained when using Procedure 4 with three faults simultaneously, compared to Procedure 3. For *s*820, the fault coverage is increased when using Procedure 4. Recycling also benefits from the processing of several faults simultaneously. We point out that the lengths of the test sequences produced by the underlying test generation procedure are often smaller than those of other test generation procedures that do not employ compaction techniques, such as [8]. The results in Tables 2(a) and (b) show the improvement achievable by recycling compared to the underlying test generation procedure embedded in the recycling process. It also demonstrates the usefulness of targeting several faults simultaneously during test generation.

Next, we consider the use of several faults to select the suffix in Procedure 2. The goal of this study is to reduce the number of iterations of Procedure 2 and its computational effort (reported below).

Table 2: Results of Procedure 2
(a) Using Procedure 3

circuit	flts	under det	under len	recycl det	recycl len	iter
s208	215	137	153	137	114	6
s298	308	265	180	265	129	2
s344	342	329	127	329	49	5
s386	384	314	247	314	185	7
s420	430	179	128	179	101	3
s641	467	403	152	404	93	9
s820	850	623	258	719	304	9

(b) Using Procedure 4 with $N_f = 3$

circuit	under det	under len	recycl det	recycl len	iter
s208	137	108	137	94	3
s298	265	169	265	115	5
s344	329	141	329	65	9
s386	314	201	314	169	9
s420	179	117	179	95	6
s641	404	153	404	153	0
s820	760	405	794	416	9

Step 3 of Procedure 2 is the following.

(3) Select a fault f_0 such that $u_{det}(f_0) = \max \{u_{det}(f): f \in F\}$.

We replace this step by a procedure that selects a subset of faults F_0 instead of the single fault with the latest detection time. The size of F_0 is bounded by a preselected constant N_0. We include in F_0 the faults with the highest detection times. However, we include a fault f in F_0 only if its detection time satisfies $u_{det}(f) \geq \max \{u_{det}(f): f \in F\} - \Delta$ where Δ is a preselected constant. For example, if the maximum detection time is 100 and $\Delta = 20$, we only consider faults with detection times 80 or above. By not allowing the detection time to be too low, we restrict the length of the selected suffixes. Procedure 5 given next summarizes the selection of the faults in F_0.

Procedure 5: Fault selection for recycling

(1) Let F be the set of yet-undetected target faults.

(2) Set $d = \max \{u_{det}(f): f \in F\}$.

(3) For every $f \in F$ such that $f \notin F_0$, if $u_{det}(f) = d$:
Add f to F_0. If $|F_0| = N_0$, stop.

(4) Set $d = d - 1$. If $d \geq \max \{u_{det}(f): f \in F\} - \Delta$, go to Step 3.

(5) Stop.

We applied Procedure 2 using $N_0 = 3$ and $\Delta = 20$. We used Procedure 4 with three faults targeted simultaneously during test generation ($N_f = 3$). The results are shown in Table 3, including the number of detected faults and the final test length before recycling; the number of detected faults and the final test length after recycling; and the number of recycling iterations. It can be seen that using $N_0 = 3$ reduces the number of recycling iterations and in some cases also helps reduce the test length compared to Table 2(b). We include in Table 3 results for additional circuits, that were not considered in Table 2(b). Larger circuits may be considered by embedding a more efficient test generation procedure into the recycling procedure.

To provide an indication of the computational effort involved in recycling, we measured the test generation time during every iteration of the procedure. We denote the test generation time at iteration i by RT_TG_i. We report in Table 4 the

Table 3: Results using $N_f = 3$ and $N_0 = 3$

circuit	flts	under det	under len	recycl det	recycl len	iter
s208	215	137	108	137	91	2
s298	308	265	169	265	109	3
s344	342	329	141	329	59	5
s382	399	364	783	364	578	2
s386	384	314	201	314	173	3
s420	430	179	117	179	105	3
s641	467	404	153	404	103	8
s820	850	760	405	794	459	6
s1196	1242	1235	957	1235	837	7
s1488	1486	1444	560	1444	458	3

value of RT_TG_i/RT_TG_0 for all the iterations. By considering relative run times instead of absolute run times, we remove the effects of the inefficiences in the implementation of the test generation procedure on the reported numbers, as well as the effects of circuit size. In addition to test generation time, there is also a fault simulation effort to extract the suffix in every iteration. We captured the total run time for iteration i in a variable RT_C_i. We report the value of RT_C_i/RT_TG_0 in the second row for each circuit in Table 4. At the end of the second row we show the value of $\sum_{i=0}^{9} RT_C_i/RT_TG_0$. From Table 4 it can be seen that for most circuits, additional recycling iterations do not take significant amounts of time compared to complete test generation (performed in iteration 0). For example, for $s208$, recycling increases the test generation time by 41%. For circuits such as $s820$ and $s1196$, where recycling incurred large increases in run time, we increased N_0, the size of F_0. We also reduced the maximum detection time of any fault included in F_0 by increasing Δ. These changes result in longer suffixes that detect more faults, leaving fewer faults to be detected by test generation, and reducing the test generation effort. The results obtained using the modified values of N_0 and Δ are marked with asterisks next to the circuit name. For $s820$, after increasing N_0 to 10 and Δ to 50, 792 detected faults and a test length of 443 were obtained (instead of 794 detected faults and a test length of 459 when $N_0 = 3$ and $\Delta = 20$). For $s1196$, after increasing N_0 to 20 and Δ to 100, 1235 faults were detected and the test length was 865 (instead of 1235 detected faults and a test length of 837 when $N_0 = 3$ and $\Delta = 20$). It can be seen that by changing the values of N_0 and Δ, it is possible to control the run time of the procedure without significantly changing the fault coverage or the compaction level. The fault coverage need not change at all if a more robust test generation procedure is embedded in the recycling process. The values of N_0 and Δ can be selected dynamically according to the number of faults left undetected by the suffix to further reduce the run time.

5. Concluding remarks

We introduced a new concept of test compaction, referred to as *recycling*. Given a test sequence T_1, the recycling operation keeps a suffix S_1 that detects one or more of the hardest-to-detect faults, and discards the rest of the sequence. The suffix S_1 is then used as a prefix of a new test sequence T_2. In this process, S_1 is expected to detect the more difficult to detect faults as well as many of the easy-to-detect faults, resulting in a new sequence T_2 which is shorter than T_1. Recycling was applied iteratively to obtain increasingly compacted test sequences. The run time of the recycling process was kept low by the fact that many of the faults, including hard-to-detect faults, are detected by the prefix of the new test sequence, and test generation needs to be carried out for a relatively small number of faults. Recycling was enhanced by a scheme where several faults were targeted simultaneously to generate the shortest possible test sequence that detects all of them.

Table 4: Relative test generation times

circuit	relative time during iteration i = 0	1	2	3	4	5	6	7	8	9	total
s208	1.00	0.25	0.05	0.01							
	1.02	0.27	0.08	0.04							1.41
s382	1.00	0.11	0.11	0.18	0.08	0.07	0.03				
	1.00	0.11	0.11	0.18	0.08	0.07	0.03				1.58
s386	1.00	0.30	0.25	0.19	0.25	0.23	0.22	0.13	0.11	0.06	
	1.01	0.31	0.26	0.20	0.27	0.24	0.23	0.14	0.12	0.08	2.86
s641	1.00	0.03	0.14	0.09	0.02	0.02	0.01	0.00	0.00	0.00	
	1.00	0.03	0.14	0.09	0.02	0.02	0.01	0.00	0.00	0.00	1.31
s820	1.00	0.57	0.64	0.47	0.59	0.52	0.50	0.56	0.42	0.45	
	1.02	0.59	0.66	0.49	0.61	0.54	0.52	0.59	0.44	0.48	5.94
s820*	1.00	0.54	0.40	0.43	0.38	0.24	0.16	0.20	0.21	0.06	
	1.02	0.56	0.42	0.45	0.41	0.27	0.18	0.23	0.24	0.09	3.87
s1196	1.00	0.79	0.73	0.54	0.54	0.55	0.53	0.47	0.42	0.41	
	1.02	0.82	0.76	0.57	0.57	0.58	0.56	0.50	0.45	0.44	6.27
s1196**	1.00	0.49	0.49	0.50	0.30	0.20	0.21	0.12	0.06	0.01	
	1.02	0.52	0.53	0.54	0.35	0.24	0.26	0.17	0.11	0.06	3.80
s1488	1.00	0.22	0.19	0.18	0.14	0.12	0.11	0.12	0.10	0.11	
	1.00	0.22	0.19	0.19	0.14	0.12	0.11	0.13	0.11	0.12	2.33

* $N_0 = 10$ and $\Delta = 50$ ** $N_0 = 20$ and $\Delta = 100$
For other circuits, $N_0 = 3$ and $\Delta = 20$

References

[1] I. Pomeranz and S. M. Reddy, "On Generating Compact Test Sequences for Synchronous Sequential Circuits" in Proc. EURO-DAC '95, Sept. 1995, pp. 105-110.

[2] S. T. Chakradhar and A. Raghunathan, "Bottleneck Removal Algorithm for Dynamic Compaction and Test Cycles Reduction", in Proc. EURO-DAC '95, Sept. 1995, pp. 98-104.

[3] I. Pomeranz and S. M. Reddy, "Dynamic Test Compaction for Synchronous Sequential Circuits using Static Compaction Techniques", in Proc. 26th Fault-Tolerant Computing Symp., June 1996, pp. 53-61.

[4] E. M. Rudnick and J. H. Patel, "Simulation-based Techniques for Dynamic Test Sequence Compaction", in Proc. Intl. Conf. on Computer-Aided Design, Nov. 1996.

[5] R. K. Roy, T. M. Niermann, J. H. Patel, J. A. Abraham and R. A. Saleh, "Compaction of ATPG-Generated Test Sequences for Sequential Circuits", in Proc. Intl. Conf. on Computer-Aided Design, Nov. 1988, pp. 382-385.

[6] B. So, "Time-Efficient Automatic Test Pattern Generation Systems", Ph.D. Thesis, EE Dept., Univ. of Wisconsin at Madison, 1994.

[7] I. Pomeranz and S. M. Reddy, "On Static Compaction of Test Sequences for Synchronous Sequential Circuits", in Proc. 33rd Design Autom. Conf., June 1996, pp. 215-220.

[8] T. Niermann and J. H. Patel, "HITEC: A Test Generation Package for Sequential Circuits", European Design Autom. Conf., 1991, pp. 214-218.

[9] T. P. Kelsey and K. K. Saluja, "Fast Test Generation for Sequential Circuits", Intl. Conf. Comp. Aided Design, Nov. 1989, pp. 354-357.

[10] J. F. McDonald and C. Benmehrez, "Test Set Reduction Using the Subscripted D-Algorithm", 1983 Intl. Test Conf., August 1983, pp. 115-121.