On Improving Genetic Optimization based Test Generation

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

Abstract

Test generation procedures based on genetic optimization were shown to be effective in achieving high fault coverage for benchmark circuits. In a genetic optimization procedure, the crossover operator accepts two test patterns t_1 and t_2, and randomly copies parts of t_1 and parts of t_2 into one or more new test patterns. Such a procedure does not take advantage of circuit properties that may aid in generating more effective test patterns. In this work, we propose a representation of test patterns where subsets of inputs are considered as indivisible entities. Using this representation, crossover copies all the values of each subset either from t_1 or from t_2. By keeping input subsets undivided, activation and propagation capabilities of t_1 and t_2 are captured and carried over to the new test patterns. The effectiveness of this scheme is demonstrated by experimental results.

1. Introduction

Test generation procedures based on genetic optimization [1] were shown to be effective in achieving high fault coverage for benchmark circuits [2]-[5]. However, as stand-alone procedures, test generation procedures based on genetic optimization may fail to achieve complete fault coverage in reasonable time. For example, in [2], the fault coverages achieved for almost all the ISCAS-85 benchmark circuits are smaller than those achievable by deterministic test generation procedures. To alleviate this problem, test generation procedures based on genetic optimization were combined with deterministic test generation procedures [6], [7]. In this work, we address one of the possible causes of this deficiency of genetic optimization.

In the context of test generation, the genetic optimization procedure starts with an initial test set (typically, a set of random patterns). At every iteration, a *fitness* value is ascribed to every member of the test set. Pairs of tests from the test set are randomly selected and used for creating new tests. The probability of selecting a test to be part of a pair is proportionate to its fitness value. The selected pairs are used to create new tests by using the genetic operators of *crossover* and *mutation*. The conventional crossover operator [1], [3] proceeds as follows. Consider two test patterns $u = [u_1 u_2 \cdots u_n]$ and $v = [v_1 v_2 \cdots v_n]$. A number c between 0 and $n-1$ is randomly selected. This number is called the *crossover point*. Two new patterns are obtained by copying the first c bits of u (v) and the last $n-c$ bits of v (u) into the same vector. The two new vectors obtained are $w_1 = [u_1 \cdots u_c v_{c+1} \cdots v_n]$ and $w_2 = [v_1 \cdots v_c u_{c+1} \cdots u_n]$. In [4], this basic crossover operator is extended to the case where two test sequences u and v are considered, by copying a randomly determined prefix of u and a randomly determined suffix of v into a new test sequence. In [3] and [5], each individual input is randomly copied from one of two sequences (*uniform crossover*). The *mutation* operator complements each bit in the new tests(s) with probability p_M. The value of p_M is typically very small. Mutation creates new solutions and explores parts of the search space that would not be explored otherwise.

Considering combinational circuits, the crossover operator is expected to combine useful information from two test patterns u and v into new test patterns that will detect faults that are not detected by either u or v. However, an arbitrary combination of values from two different tests may not be useful in detecting new faults. The crossover procedure in [2] attempts to alleviate this problem by taking test generation objectives into account. Given two test vectors u and v, the procedure first identifies input values in u and v that are useful in creating circuit activity in subcircuits that currently have low levels of activity. Activity is measured by the number of events created during logic simulation of a vector. An input value that creates activity is identified by complementing it and simulating the resulting vector. Input values that create activity are copied from the two vectors u and v to create a new vector. This crossover procedure has the following disadvantages. (1) It relies on consideration of single input values, whereas fault detection may require to simultaneously assign specific values to specific inputs. (2) Circuit activity is not an accurate measure of fault detection, and its determination requires logic simulation.

To alleviate the shortcomings of the crossover operators used in [2]-[5], we propose the following approach. We find subsets of inputs that determine the values of internal lines in the circuit. In test patterns that detect one or more faults, such subsets of inputs contribute to the activation and/or propagation of faults within the subcircuits they affect. During crossover, we consider these subsets as whole entities, and we copy the values of all the inputs in a subset together into new test patterns. The new patterns created in this way maintain the activation and propagation capabilities of the original patterns, and are likely to detect additional faults. In addition, no logic or fault simulation is required during crossover. To comply with the genetic algorithm terminology [13], the proposed approach can be viewed as a representation scheme, as follows. Conventionally, a test pattern is represented as a bit vector. In the proposed scheme, a test pattern is defined over precomputed subsets of inputs. For example, consider a four-input circuit with inputs x_1, x_2, x_3, x_4. Let the input subsets be $\{x_1, x_2, x_3\}$ and $\{x_3, x_4\}$ (note that the subsets are overlapping). Then the test pattern conventionally written as (0011) is viewed under the proposed scheme as the pattern (001)(11), where the inputs $(x_1 x_2 x_3)$ assume the pattern

+ Research supported in part by NSF Grant No. MIP-9220549, and in part by NSF Grant No. MIP-9357581

ED&TC '97 on CD-ROM

(001) and the inputs ($x_3 x_4$) assume the pattern (11).

The proposed representation scheme is embedded into a test generation procedure for combinational circuits that uses uniform crossover and mutation to generate new test patterns, starting from randomly selected ones. The input subsets are computed in a preprocessing step. Alternatively, it is possible to compute them dynamically during the test generation process. For comparison purposes, the conventional representation scheme of test patterns as bit vectors using several crossover operators [1], [3], [5] are also embedded into the same test generation procedure. Comparison of the results demonstrates the advantages of the proposed scheme in terms of fault coverage and/or the number of iterations required to achieve the fault coverage. Preliminary results for synchronous sequential circuits are also presented. We expect that embedding of the proposed representation scheme into the test generation procedures of [3]-[5] would result in similar improvements.

The paper is organized as follows. In Section 2 we show an example to motivate the proposed scheme. Section 3 describes the selection of input subsets. In Section 4 we describe the test generation procedure for combinational circuits. In Section 5 we present experimental results of the proposed procedure and compare it to several other schemes. In Section 6 we show preliminary results for synchronous sequential circuits. Section 7 includes concluding remarks.

2. An example

For illustration, we use the circuit of Figure 1. In this circuit, the following subsets of inputs determine the values of internal lines. The subset of inputs {1,2} determines the value of line 9; the subset of inputs {3,4} determines the value of line 10; the subset of inputs {5,6} determines the value of line 11; and the subset of inputs {7,8} determines the value of line 12. Thus, these input subsets may be effective in activating and propagating faults. Given two tests that detect some faults in the circuit, we would like to maintain the values of these input subsets and create new tests that include them. For example, the test t_1 = (11101101) detects the faults 9 s.a. 0 and 11 s.a. 0. The test t_2 = (01111011) detects the faults 10 s.a. 0 and 12 s.a. 0. If we take the values of inputs {1,2} from t_2, the values of inputs {3,4} from t_1, the values of inputs {4,5} from t_2, and the values of inputs {6,7} from t_1, we obtain the pattern t = (01101001), that detects the faults 1 s.a. 1, 4 s.a. 1, 6 s.a. 1, 7 s.a. 1, 13 s.a. 1 and 14 s.a. 1. Thus, by maintaining the values of the pairs of inputs above intact, we can detect six additional faults after crossover. The conventional representation scheme may not readily lead to such a new test.

3. Selecting input subsets

In this section, we describe the selection of the input subsets whose values will be manipulated as single entities during genetic optimization.

To ensure that input subsets are effective in activating and propagating faults in the circuit, we select the input subsets such that each one of them is the input cone of some line in the circuit. The *input cone c* of line *g* is the set of all the primary inputs that drive line *g*. We exclude from consideration input cones of primary inputs since they have size 1; input cones of fanout branches since they are the same as the cones of the fanout stems; and input cones of outputs of single-input gates since they are the same as the cones of the gate inputs. To illustrate the procedure for selecting the input cones we use ISCAS-85 benchmark circuit *c*17, shown in Figure 2. The input

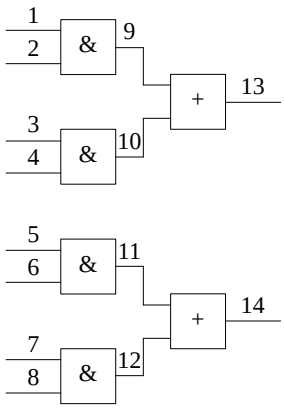

Figure 1: An example circuit

cones of all the lines in the circuit excluding primary inputs and fanout branches are shown in Table 1.

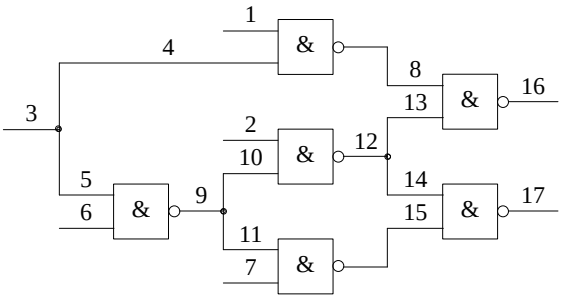

Figure 2: Circuit *c*17

Table 1: The input cones of *c*17

line	input cone	line	input cone
8	1,3	15	3,6,7
9	3,6	16	1,2,3,6
12	2,3,6	17	2,3,6,7

Several considerations are used in selecting the input cones (or input subsets) for the proposed representation scheme. (1) Regarding the sizes of the input subsets, we would like the input subsets to be large enough to capture activation and propagation information. At the same time, we would like the input subsets to be small enough to allow a large variation in the generated test patterns due to crossover. In the proposed procedure, we use a parameter $N_{PI,max}$ to determine the maximum size of the input subsets considered. We set $N_{PI,max}$ to be as low as possible, but not lower than another constant $N_{PI,INIT}$. In our experiments, $N_{PI,INIT}$ = 10. (2) We require that every primary input of the circuit would participate in at least one input subset. This ensures that when a new test is generated, every one of its input values is determined. (3) We allow an input to participate in more than one cone. This is required since in some cases, it is impossible to avoid overlapping input cones. Crossover using overlapping input subsets is considered in Section 4.

The procedure for selecting the input subsets proceeds as follows. We set an upper bound $N_{PI,max}$ on the size of an input cone that will be considered. The value of $N_{PI,max}$ is initialized to a constant $N_{PI,INIT}$, and it is increased until a subset of input cones can be selected, that includes all the inputs. For a given value of $N_{PI,max}$, we execute Procedure 1 shown in Figure 3 to select a set of input cones. In Procedure 1, we assume that the

lines in the circuit are numbered $1, 2, \cdots, L$ in increasing order from inputs to outputs. The lines in the circuit are considered from primary outputs to primary inputs. The cone of line g is selected by Procedure 1 if its size does not exceed $N_{PI,max}$. Suppose that the cone of line g is selected. Let h be a line that drives g. Then the input cone of h is contained in the input cone of g, and h is not considered by Procedure 1. Of the cones selected in this process, some may be redundant. A cone is said to be *redundant* if all its inputs are also included in other cones. We use a greedy procedure in Step 4 of Procedure 1 to find an irredundant cover C_{irr} of the set of cones C produced in Steps 1-3. For $c17$ with $N_{PI,max} = 3$, Procedure 1 selects the set of cones {{3,6,7}, {2,3,6}, {1,3}} belonging to lines 15, 12 and 8, respectively. All the primary inputs are included in these cones. None of the cones is redundant.

Procedure 1: Finding input cones for $N_{PI,max}$

(1) Unmark every line in the circuit. Mark the primary outputs. Set $g = L$. Set $C = \phi$.
(2) If (a) g is a primary output or the output of a multi-input gate, (b) g is marked, and (c) the size of the input cone of g does not exceed $N_{PI,max}$:
Add the cone of line g to the set of cones C.
Else, mark all the lines that immediately drive line g (line h is said to *immediately drive* line g if h is an input of a gate with output g, or if h is a fanout stem with a branch g).
(3) Set $g = g - 1$. If $g \geq 1$ go to Step 2.
(4) Find an irredundant cover C_{irr} of C.
(5) Return C_{irr}.

Figure 3: Procedure 1

Procedure 1 is called repeatedly with increasing values of $N_{PI,max}$, until for every primary input I, there is a cone $c \in C$ such that $I \in c$. The following observations are used to shorten the search for an appropriate value of $N_{PI,max}$.

Consider the set of all input cones C_{all} of all the lines in the circuit excluding primary inputs, fanout branches and outputs of single-input gates (e.g., Table 1 shows C_{all} for $c17$). For each primary input I, let $N_{I,min}$ be the number of inputs in the smallest cone that includes input I. To allow input I to be included in some cone in C, we must have $N_{PI,max} \geq N_{I,min}$. We can thus start with $N_{PI,max} = \max\{N_{PI,INIT}, N_{I,min}$: I *is a primary input*$\}$ ($N_{PI,INIT}$ was defined to ensure that the cones are not too small). For example, for $c17$ with $N_{PI,INIT} = 1$, $N_{PI,max} = 3$ since the smallest cone that includes input 2 (7) is of size 3 (cf. Table 1).

Next, consider the case where the set of cones C selected for $N_{PI,max}$ does not include all the primary inputs. We observe that unless we increase $N_{PI,max}$ sufficiently to allow new cones to be included in C, that were not included based on the previous value of $N_{PI,max}$, we will obtain the same set of cones again. Let g be the line with the smallest cone c that is larger than $N_{PI,max}$. We increase $N_{PI,max}$ to the size of c. The complete procedure for finding the set of cones is shown in Figure 4.

4. Test generation

The test generation procedure is described in this section and summarized in Procedure 3 shown in Figure 5. In the proposed procedure, a single test is an *individual* in the genetic optimization terminology. All the tests together form the *population*. The tests that detect any new faults are carried over from one iteration (or *generation*) of the genetic optimization procedure to the next. Thus, we have a test set T that grows from one iteration to the next by adding new tests that detect yet-undetected faults. Only fault dropping simulation is performed to determine the effectiveness of a test (its *fitness function* value). The fitness function value of a test t is equal to the number of yet-undetected faults detected by t when it is first entered into the population (the fitness of a test can also be determined by using non-fault dropping simulation, however, this would require larger simulation times). In addition, each test in the population is simulated only once throughout the iterations. This is accomplished by maintaining a single fault list that initially contains all the target faults, and dropping detected faults from this list every time a new test is added to the population. Since the fitness values tend to decrease as more tests are produced and the fault list shrinks, we distinguish between three levels of fitness (instead of considering specific fitness values). Tests that do not detect any new faults when they are generated are given a fitness value of 0, and are not used for producing new tests. Tests that detect large numbers of faults are selected to create new tests with twice the probability of tests that detect low numbers of faults. This selection process is explained below.

Procedure 2: Finding input cones for test generation

(1) Let $N_{PI,max} = N_{PI,INIT}$.
(2) For each primary input I:
(a) Let $N_{I,min}$ be the number of inputs in the smallest cone that includes input I.
(b) If $N_{I,min} > N_{PI,max}$, set $N_{PI,max} = N_{I,min}$.
(3) Call Procedure 1 with $N_{PI,max}$ to obtain a set of cones C.
(4) If, for every primary input I, there exists a cone $c \in C$ such that $I \in c$, return C.
(5) Let g be the line with the smallest cone c that is larger than $N_{PI,max}$. Set $N_{PI,max}$ equal to the size of c.
(6) Go to Step 3.

Figure 4: Procedure 2

Test generation starts by creating an initial test set. The initial test set is created by simulating a constant number $N_{T,MAX}$ of random patterns. The patterns effective in detecting any yet-undetected faults are kept in a test set T. The other patterns are discarded. Eliminating patterns that do not detect any new faults is equivalent to giving them a fitness function value of 0, and not selecting them for crossover to create new tests. The initial test set T is generated in Step 2 of Procedure 3. Several iterations follow, where new patterns are generated based on the existing patterns in T. Each new pattern is simulated, and if it detects any new faults, it is kept in T. At most $N_{T,MAX}$ patterns are generated, or until the size of T reaches $N_{T,MAX}$. Generation of new patterns is controlled by Steps 3-6 of Procedure 3.

New patterns are generated in Procedure *gen_test*(). The procedure uses uniform crossover over the proposed representation of test patterns. The procedure accepts the current test set T, and the following subset. Let the test set T before the beginning of the current iteration be T'. We define a subset T'_{best} of T', that contains the tests that detect the larger numbers of faults in T'. Tests are included in T'_{best} starting from the tests that detect the largest number of faults, followed by the tests that detect the second-to-maximum number of faults, and so on, until the size of T'_{best} is at least $T'_{best,MIN}$. In our experiments, $T'_{best,MIN} = T'/3$. New patterns are created using two tests t_1, t_2. The tests t_1 and t_2 are randomly selected either out of T or out of T'_{best}, as follows. First, either T'_{best} or T is selected with probability $\frac{1}{2}$. Then,

two tests are randomly selected out of the selected set (T'_{best} or T). Since $T'_{best} \subset T$, the tests in T'_{best} have twice the probability of being selected over the tests in $T - T'_{best}$. Thus, this selection scheme gives higher probability to the selection of tests that detect larger numbers of faults, yet it allows every test in T to be used with non-zero probability. The selection of t_1 and t_2 is done in Steps 1 and 2 of Procedure *gen_test*().

A new test t is created by selecting some input cone values out of t_1 and some values out of t_2. The source of the values is determined by the variables { $from\_test(c)$: $c \in C$} in Step 3 of Procedure *gen_test*(). For each input cone c, we randomly determine whether its values will come from t_1 or from t_2. The values are assigned into t in a random order (Steps 4 and 5 of Procedure *gen_test*()), for the following reason. Consider a circuit with 5 inputs $\{x_1, x_2, x_3, x_4, x_5\}$ and two cones, c_1 and c_2, with inputs $\{x_1, x_2, x_3\}$ and $\{x_1, x_4, x_5\}$, respectively. Let t_1 = (00000) and let t_2 = (11111). Let $from\_test(c_1) = t_1$ and let $from\_test(c_2) = t_2$. If we assign the values to t in the order $<c_1, c_2>$, we first assign 0s to x_1, x_2, x_3, and then assign 1s to x_1, x_4, x_5. The result is the pattern (10011). If we assign the values to t in the order $<c_2, c_1>$, we obtain the pattern (00011). We obtained two different patterns, depending on the order in which the input cone values were assigned. This is a consequence of the overlap between the subsets of inputs c_1 and c_2. To avoid giving preference to the input values of certain cones, we randomly permute them before determining t.

The probability of mutation in our implementation of the genetic optimization procedure is $\frac{1}{N_{PI}}$, where N_{PI} is the number of circuit inputs. Probability values that are inversely proportionate to the number of bits in a solution were found to be effective in [8]. Mutation is done in Step 6 of Procedure *gen_test*().

Note that the value of $N_{T,MAX}$ in Procedure 3 must be larger than the minimum test set size, otherwise, after finding $N_{T,MAX}$ tests that detect any faults, no additional patterns will be considered. We experimented with various values of $N_{T,MAX}$ as reported in the next section. It is also possible to dynamically increase the test set size beyond $N_{T,MAX}$ if it contains close to $N_{T,MAX}$ tests. We did not explore this option.

5. Experimental results

Experimental results of the proposed procedures are reported in this section for ISCAS-85 benchmark circuits [9]. The circuit *c*6288 is omitted since it is testable by a small number of pure-random patterns.

Information about the input cones generated by Procedure 2 is shown in Table 2. After circuit name we show the number of primary inputs, the value of $N_{PI,max}$ for which a set of cones was found, and the number of cones after removing redundant cones. Note that $N_{PI,max}$ is the maximum number of inputs in a cone, and that there may be a large number of cones with smaller numbers of inputs. For example, in *c*432, we have (after removing redundant cones) one cone with 18 inputs, and 18 cones with two inputs each. Thus, there is overlap between the inputs in different cones. Procedure 3 was designed to handle such overlap.

Results of the proposed test generation procedure using the cones reported in Table 2 are also shown in Table 2. We used $N_{T,MAX} = 1,000$ in this experiment and allowed at most 1,000 iterations. Under column *det.able* we show the number of detectable stuck-at faults in each circuit. Under column *det.ed*

Procedure 3: Test generation

(1) Let F be the set of target faults. Let $T = \phi$. Let C be the set of cones produced by Procedure 2.
(2) Repeat $N_{T,MAX}$ times:
  (a) Generate a random pattern t.
  (b) Simulate t. Let $F(t)$ be the set of faults detected by t.
  (c) If $|F(t)| > 0$:
    (i) Add t to T.
    (ii) Set $F = F - F(t)$.
(3) Set $N_{iter} = 1$.
(4) Set $T' = T$. Let T'_{best} contain at least $T'_{best,MIN}$ tests out of T that detect the larger numbers of faults.
(5) Repeat $N_{T,MAX}$ times, or until $|T| = N_{T,MAX}$:
  (a) Generate a new test, t, by calling procedure $gen\_test(T, T'_{best})$.
  (b) Simulate t. Let $F(t)$ be the set of faults detected by t.
  (c) If $|F(t)| > 0$:
    (i) Add t to T.
    (ii) Set $F = F - F(t)$.
(6) Set $N_{iter} = N_{iter} + 1$. If $N_{iter} \leq N_{iter,MAX}$ and $F \neq \phi$, go to Step 4.

Procedure $gen\_test(T, T'_{best})$**:**

(1) Let $type_1$ be a random number out of {0,1}.
(2) If $type_1 = 0$, select two tests t_1 and t_2 randomly out of T'_{best}.
  Else, select two tests t_1 and t_2 randomly out of T.
(3) For every cone c:
  (a) Let $type_2$ be a random number out of {0,1}.
  (b) If $type_2 = 0$, set $from\_test(c) = t_1$.
    Else, set $from\_test(c) = t_2$.
(4) Let $cone\_order = < c_{i1}, c_{i2}, \cdots, c_{iN_C} >$ be a random permutation of the input cones in C.
(5) For $j = 1, 2, \cdots, N_C$:
    Copy the input values of cone c_{ij} in test $from\_test(c_{ij})$ into t.
(6) For every primary input i:
    Select a random number *compl* out of $\{0, 1, \cdots, N_{PI} - 1\}$. If $compl = 0$, complement the value of input i in t.
(7) Return t.

Figure 5: Procedure 3

we show the number of faults detected by the proposed procedure. The number of patterns simulated by Procedure 3 before the final test set was obtained is shown under column *patt*. The number of simulated patterns is computed as follows. If Procedure 3 goes through n iterations, simulating $N_{T,MAX}$ patterns at every iteration, then the total number of simulated patterns is not larger than $(n+1)N_{T,MAX}$ (including the construction of the initial test set). Note that the final fault coverage may be reached before an iteration is completed, however, we count the number of simulated patterns in multiples of $N_{T,MAX}$. Under column *tests* we show size of the test set derived. It may be possible to further compact the test set, e.g., by reverse order fault simulation. It can be seen that the proposed procedure detects all the detectable faults in all the circuits.

In Table 3, we compare the proposed test generation procedure to several other procedures. The number of faults detected in [2] is shown under column [2] of Table 3. Next, we

Table 2: Results using $N_{T,MAX} = 1,000$

circuit	inp	$N_{PI,max}$	cones	det.able	det.ed	patt	tests
c432	36	18	19	520	520	2000	76
c499	41	10	8	750	750	2000	67
c880	60	18	16	942	942	3000	115
c1355	41	10	8	1566	1566	3000	109
c1908	33	15	8	1870	1870	3000	176
c2670	233	14	108	2630	2630	138000	187
c3540	22	22	6	3291	3291	24000	276
c5315	178	17	91	5291	5291	4000	221
c7552	207	21	25	7419	7419	787000	389

show the number of faults detected by applying up to 1,000,000 pure-random patterns (this is also the number of patterns simulated by the proposed procedure in 1,000 iterations). We also show the number of patterns that needed to be applied before the reported number of detected faults was reached. For example, for *c*432, 520 faults were detected after 1,190 random patterns were applied. Under column *conv.*1*point* we show the results of test generation using Procedure 3, except that the conventional representation is used and the crossover operator uses a single crossover point. Under column *conv.uni* we show the results of test generation using Procedure 3, except that the conventional representation with uniform crossover are used. This implies that each input value in a new test is randomly selected out of one of two tests.

Table 3: Comparison with other procedures

		pure random		conv.1point		conv.uni	
circuit	[2]	det.ed	patt	det.ed	patt	det.ed	patt
c432	519	520	1190	520	2000	520	2000
c499	749	750	1159	750	2000	750	2000
c880	937	942	10102	942	3000	942	3000
c1355	1556	1566	2529	1566	3000	1566	3000
c1908	1852	1870	6742	1870	5000	1870	4000
c2670	2290	2357	128000	2630	827000	2630	360000
c3540	3277	3291	37686	3291	96000	3291	40000
c5315	5258	5291	3934	5291	4000	5291	4000
c7552	7120	7270	128000	7366	587000	7376	699000

It can be seen that in most cases, the fault coverage achieved by the proposed procedure is higher than the fault coverage achieved by the procedure of [2]. Circuits *c*432, *c*499, *c*1355 and *c*5315 are testable by small numbers of random patterns, and the genetic optimization procedures do not perform better than pure random patterns. For *c*880 and *c*1908, fewer patterns are simulated by using the genetic optimization procedures than when pure random patterns are used. All three genetic optimization procedures perform the same for *c*880. For *c*1908, the proposed procedure requires the smallest number of iterations. Circuit *c*3540 is random pattern testable by approximately 37,000 patterns. The genetic optimization procedures based on conventional representation require more patterns to detect all the faults; however, the proposed procedure requires only 24,000 patterns to be simulated. Circuits *c*2670 and *c*7552 are not random pattern testable. After 1,000,000 random patterns, not all the faults are detected. For *c*2670, the proposed procedure requires less than half the number of patterns than when the conventional representation and uniform crossover are used, and less than a fifth of the number of patterns when conventional representation and single point crossover are used. All three procedures detect all the faults within 1,000 iterations. For *c*7552, the two genetic procedures with the conventional representation did not detect all detectable faults after 1,000 iterations (no additional faults were detected after iterations 586 and 698). The proposed procedure detected all the faults after simulating 787,000 patterns (or 786 iterations).

To capture variations in the results due to the initial random set of $N_{T,MAX}$ tests, we repeated 10 times the test generation process for *c*2670. In each of the 10 runs we used $N_{T,MAX} = 1,000$, but we started the random pattern generation for the initial test set from a different seed. The proposed procedure achieves complete fault coverage in all 10 runs, whereas the procedure with uniform crossover achieves complete fault coverage in 9 out of 10 cases. In all but one case, the proposed procedure requires simulation of fewer patterns to achieve complete fault coverage. On the average, the number of patterns simulated by the proposed procedure is three times smaller than that of the procedure with uniform crossover.

Next, we checked the effect of varying $N_{T,MAX}$ on the fault coverage obtained by Procedure 3, using uniform crossover with the conventional representation and with the proposed representation. We considered $N_{T,MAX} = 200$ using 10 different seeds for the initial random pattern generation. The results for *c*2670 showed that uniform crossover did not result in complete fault coverage for any seed, whereas the proposed procedure resulted in complete fault coverage for 9 out of 10 seeds. The results reported above clearly indicate the superiority of the proposed representation compared to the conventional one.

The goal of the next experiment is to demonstrate that the proposed test generation procedure may generate tests to detect faults that are aborted by a deterministic test generation procedure. For this purpose, we considered the circuit SQR from [10]. The circuit has 12 inputs and 6 outputs. It has 1750 faults in its collapsed fault list. The deterministic test generation procedure from [11] fails to generate complete n-detection test sets for $n \geq 6$. An n-detection test set is one that detects each target stuck-at fault n times, by n different tests. If a fault has $m < n$ different tests, a complete n-detection test set should contain all m tests for the fault. The importance of n-detection test sets is that they allow the stuck-at fault model to be used in generating tests with high defect coverages [12]. Information about the number of detectable faults in the circuit *SQR* for n between 1 and 10 is shown in Table 4, as follows. Each row corresponds to a different value of n. Following n we show the number of faults that have at least n different tests. For example, from the entry for $n = 5$, 1628 faults each has at least five tests. Next, we show the total number of fault detections if each fault is detected by either n tests or by the maximum number of tests if the fault has fewer than n tests. For example, for $n = 1$, we have 1630 fault detections; for $n = 2$, we have 1629 faults that can be detected twice (contributing $1629 \cdot 2 = 3258$ to the number of detections), and one additional fault that can be detected only once, for a total of 3259 detections. In general, if k_i faults have i tests, $1 \leq i \leq n$, then the number of detections of an n-detection test set is $\sum_{i=1}^{n} k_i$. Under column *detections GA* we show the number of detections by an extension of the proposed genetic optimization based procedure that generates n-detection test sets. This extension maintains new tests that detect faults which are not yet detected n times. The number of fault detections by the test generation procedure of [11] is shown in the last column. It can be seen that the proposed genetic optimization procedure detects every detectable fault the required number of times. The procedure of [11] aborts on several faults for $n \geq 6$.

Table 4: Results for *SQR*

n	detectable		detections GA	[11]
1	1630	1630	1630	1630
2	1629	3259	3259	3259
3	1629	4888	4888	4888
4	1629	6517	6517	6517
5	1628	8145	8145	8145
6	1628	9773	9773	9768
7	1626	11399	11399	11388
8	1618	13017	13017	13003
9	1615	14632	14632	14603
10	1615	16247	16247	16209

6. Preliminary results for sequential circuits

The proposed representation was incorporated into a test generation procedure based on genetic optimization for synchronous sequential circuits. The procedure is different from the procedure for combinational circuits in the following points.

(1) Input cones are defined only over the primary inputs of the circuit, i.e., present-state variables that affect an internal line g are omitted from the input cone of g. (2) The test sequence length is increased during the test generation procedure as follows. The procedure starts with random test sequences of length 5. At the end of an iteration where no new faults are detected, the length of all the test sequences currently in the population is extended by adding 5 randomly selected vectors. Test length is increased at most up to 200. (3) New test sequences and extended test sequences are simulated only over the yet-undetected faults. Thus, the number of faults detected by each sequence can only increase, and never decrease even if the faults it detects are detected by other sequences in the population. (4) When crossover is performed between two test sequences T_1 and T_2, the values of each cone are randomly copied either from T_1 or from T_2 independently for each vector. (5) At the end of each iteration, new input sequences are randomly generated and added to the population if they detect any new faults. This is required to ensure that the population is rich enough to support effective evolution of new solutions. (6) The initial population is created by selecting effective test sequences out of 100 randomly generated sequences. The total number of test sequences per iteration (the population size) is also limited to 100. If the number of test sequences reaches 100 in a given iteration, it is reduced at the end of the iteration by performing the following procedures. (i) Reverse order fault simulation is used to drop test sequences that do not detect any faults beyond the ones detected by the test sequences appearing after them in the population. (ii) The remaining test sequences are simulated by decreasing order of the number of faults they detect. Sequences that do not detect any new faults are dropped.

In Table 5 we show some preliminary results we obtained using this procedure for ISCAS89 benchmark circuits. We show the number of iterations and the number of detected faults by the proposed procedure. The numbers of detected faults reported in [2] and [3] are also included for comparison ([5] uses reset and later procedures combine genetic optimization with deterministic test generation; thus, no direct comparison is possible).

7. Concluding remarks

We described a representation scheme suitable for test generation based on genetic optimization. The proposed scheme considers subsets of inputs as indivisible entities, and copies all the values of each subset from a single test. The input subsets were selected based on structural preprocessing of the circuit, where input cones of internal lines in the circuit were used to define candidate input subsets. This scheme was based on the observation that maintaining subsets of values that determine values of internal lines helps maintain activation and propagation properties that may be useful in detecting new faults. The effectiveness of this scheme was demonstrated by incorporating it into a generic procedure based on genetic optimization. With the proposed scheme, complete fault coverage was achieved for all ISCAS-85 benchmark circuits. Preliminary results of the incorporation of this scheme into a test generation procedure for synchronous sequential circuits were also presented.

Table 5: Results for synchronous sequential circuits

circuit	proposed iter	proposed det.ed	det.ed [2]	det.ed [3]
s386	68	309	292	295
s420	28	155	145	NA
s1238	61	1282	1229	1274

References

[1] J. H. Holland, *Adaptation in Natural and Artificial Systems*, University of Michigan Press, 1975.

[2] D. G. Saab, Y. G. Saab and J. A. Abraham, "CRIS: A Test Cultivation Program for Sequential VLSI Circuits", Intl. Conf. Computer-Aided Design, Nov. 1992, pp. 216-219.

[3] E. M. Rudnick, J. H. Patel, G. S. Greenstein and T. M. Niermann, "Sequential Circuit Test Generation in a Genetic Algorithm Framework", in Proc. Design Autom. Conf., June 1994, pp. 698-704.

[4] P. Prinetto, M. Rebaudengo and M. Sonza Reorda, "An Automatic Test Pattern Generator for Large Sequential Circuits based on Genetic Algorithms", in Proc. 1994 Intl. Test Conf., Oct. 1994, pp. 240-249.

[5] F. Corno, P. Prinetto, M. Rebaudengo, M. Sonza Reorda and R. Mosca, "Advanced Techniques for GA-based Sequential ATPGs", in Proc. 1996 Europ. Design & Test Conf., March 1996, pp. 375-379.

[6] D. G. Saab, Y. G. Saab and J. A. Abraham, "Iterative [Simulation-Based Genetics + Deterministic Techniques] = Complete ATPG", in Proc. 1994 Intl. Conf. on Computer-Aided Design, Nov. 1993, pp. 40-43.

[7] E. M. Rudnick and J. H. Patel, "Combining Deterministic and Genetic Approaches for Sequential Circuit Test Generation", in Proc. 32rd Design Autom. Conf., June 1995, pp. 183-188.

[8] V. Kommu, "Enhanced Genetic Algorithms in Constrained Search Spaces with Emphasis on Parallel Environments", Ph.D Thesis, Dept. of Electrical and Computer Engineering, University of Iowa, June 1993.

[9] F. Brglez and H. Fujiwara, "A Neutral Netlist of 10 Combinational Benchmark Design and a Special Translator in Fortran", in Proc. 1985 Intl. Symp. on Circuits and Systems, June 1985.

[10] P. Franco, W. D. Farwell, R. L. Stokes and E. J. McCluskey, "An Experimental Chip to Evaluate Test Techniques Chip and Experiment Design", in Proc. 1995 Intl. Test Conf., Oct. 1995, pp. 653-662.

[11] S. M. Reddy, I. Pomeranz and S. Kajihara, "On the Effects of Test Compaction on Defect Coverage", in Proc. 14th VLSI Test Symp., April 1996, pp. 430-435.

[12] S. C. Ma, P. Franco and E. J. McCluskey, "An Experimental Chip to Evaluate Test Techniques Experiment Results", in Proc. 1995 Intl. Test Conf., Oct. 1995, pp. 663-672.

[13] J. H. Patel, Private Communication.