Evolutionary test generation methods for digital devices

Y.A.Skobtsov.,V.Yu.Skobtsov

IN: DESIGN OF DIGITAL SYSTEMS AND DEVICES (EDS. M.ADAMSKI ET AL.). – BERLIN HEIDELBERG: SPRINGER VERLAG. - LECTURE NOTES IN ELECTRICAL ENGINEERING, 2011, VOL. 79, - P.331-361

13 Evolutionary test generation methods for digital devices

In this chapter, we will discuss how evolutionary methods can be used for test generation of digital circuits. In present time it is strongly investigated the new direction in theory and practice of artificial intelligence and information systems – evolutionary computations. This term is used to generic description of the search, optimizing or learning algorithms, based on some formal principles of natural evolutional selection, which are sufficiently applied in solving various problems of machine learning, data mining, databases etc [1]. Among this approaches following main paradigms can be picked out: *genetic algorithms* (GA), *evolutionary strategy* (ES), *evolutional programming* (EP), *genetic programming* (GP).

The differences of these approaches mainly consist in the way of target solution representation and in different set of evolutional operators used in evolutional simulation. Classical GA uses the binary encoding of problem solution and basic genetic operators are crossover and mutation. In ES solution is represented by real numbers vector and basic operator is mutation. EP uses FSM as solution representation and mutation operator. In GP problem solution is represented by program, crossover and mutation operators are applied. Now this classification is enough relative and interaction of basic evolutionary paradigms each other takes place.

13.1 Genetic algorithms and their modifications

GA are random directed search algorithms, which emulate natural evolution process, to construct (sub)optimal solution of given problem. Any solution is represented with a chromosome or individual – string of elements (genes). Classical "simple" GA [2] uses binary strings (for example, 0011101) to represent an individual. Therefore it looks very attractive to use GA techniques for a decision of ATPG problems for DS at structural or functional description levels. On the solution set the fitness (goal) function is determined. Fitness function allows to evaluate the closeness of each individual to the optimal solution – the ability of survival. Classical “simple” GA uses three basic operators: reproduction, crossover and mutation. Using these operators, the population (the set of individuals-solutions of considered problem) evolves from one generation to another. Classical steady state GA may be represented as the following sequence of operations that is shown in flow chart of Fig.13.1. Here parent individuals are selected with best fitness values. Then crossover is performed with a high probability P_c. The formed offspring are mutated with a low probability P_m and inserted in current population. To maintain the steady individuals number the population reducing is performed.

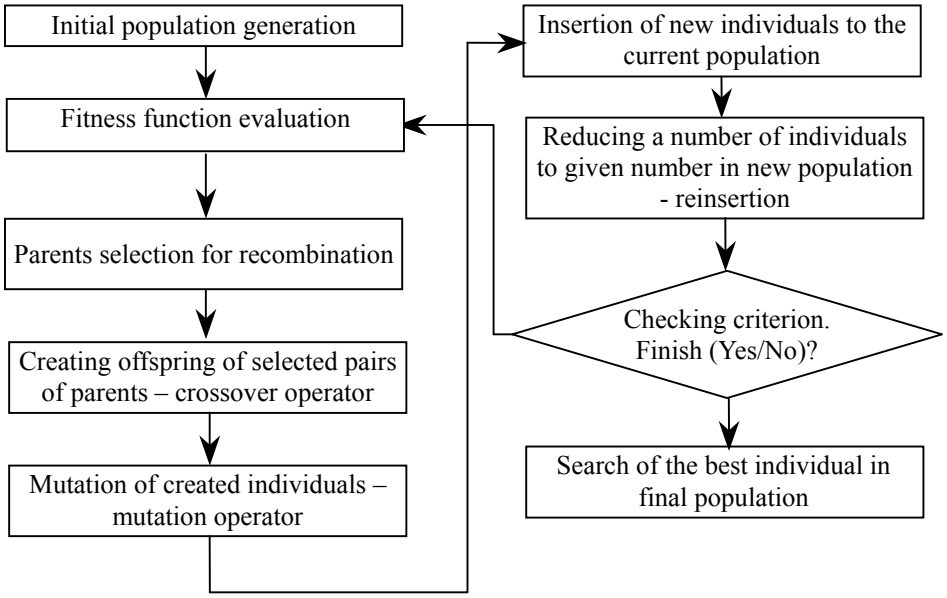


Fig.13.1 Classical "simple" GA flowchart

At present there were suggested numerous modifications and generalizations of GA: 1) new variants of each GA step implementations (Fig.13.1); 2) essential modification of algorithm structure[3]. Here we can mark up different methods of parent selection, population reduction. Different genetic operators of crossover, of mutation. Further we shall briefly consider different variants of every GA step implementation and generalization of GA.

13.1.1 Parents selection

At this step the individuals producing offspring are selected. The first step is fitness assignment. Each individual in the selection pool receives a reproduction probability depending on the own objective value and the objective value of all other individuals in the selection pool. This fitness is used for the actual selection step afterwards.

In selection the individuals producing offspring are chosen. As the result of selection intermediate population $\widetilde{P}^t$ from current population P^t (t – generation number) is generated: $P^t \to \widetilde{P}^t$. Selection operator is based on fitness function values. Various selection methods is used fitness value information differently and significantly influences on GA effectiveness. Each individual a_i^t in the selection

pool receives a reproduction probability $P_s(a_i^t)$ depending on the own fitness value and the fitness value of all other individuals in the population. And selection of individual a_i^t from current population P^t to intermediate population $\widetilde{P}^t$ is executed basing on the probability $P_s(a_i^t)$. The calculation methods of the probability $P_s(a_i^t)$ determines different selection methods:
roulette wheel selection [3]

$$P_s(a_i^t) = \frac{f(a_i^t)}{\sum_{j=1}^{N} f(a_j^t)}, \tag{13.1}$$

where $f(a_i^t)$ - fitness function value, N – population size;
linear rank-based selection [BH91] , [3]

$$P_s(a_i^t) = \frac{1}{N}(\alpha - (\alpha - \beta)\frac{i-1}{N-1}), \tag{13.2}$$

where $1 \le \alpha \le 2$ is chosen randomly, $\beta = 2 - \alpha$.

On *tournament selection* [3] m individuals are chosen randomly, then the best of them is selected as parents. This procedure is continued until intermediate population $\widetilde{P}^t$ is not formed. Here selection parameter is $2 \le m \le N$.

13.1.2 Crossover operators

Once the parents are selected, the crossover operator is used to generate offspring with a high probability P_c. The basic genetic operators and their properties can now be explained. In *single-point crossover* one crossover position $k \in \{1,2,...,L-1\}$, L – length of an individual, is selected uniformly at random and the substrings exchanged between the individuals about this point, then two new offspring are produced (Fig.13.2).

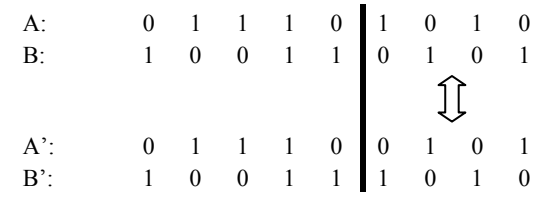


Fig.13.2 Single-point binary crossover

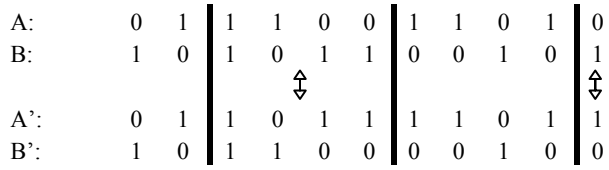


Fig.13.3 Multi-point binary crossover

For *multi-point crossover*, m crossover positions $k_i \in \{1,2,...,L-1\}$, $i = \overline{1,m}$, are chosen at random with no duplicates and sorted in ascending order. Then, the substrings between successive crossover points are exchanged between the two parents to produce two new offspring (Fig.13.3). The section between the first variable and the first crossover point is not exchanged between individuals [2,3].

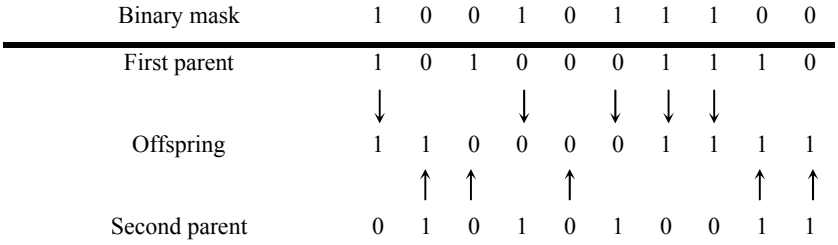


Fig.13.4 Uniform crossover

Uniform crossover [2,3] makes every locus a potential crossover point. A crossover mask, the same length as the individual structure is created at random and the parity of the bits in the mask indicate which parent will supply the offspring with which bits (for example, 1 – first parent, 0 –second parent – Fig.13.4).

13.1.3 Mutation

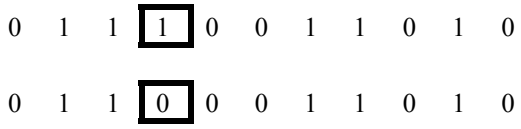


Fig.13.5 Binary mutation

As new offspring are generated, each gene is mutated with low probability P_m. Usually the probability of mutating a gene is set to be inversely proportional to the number of genes in chromosome (dimensions). The more dimensions one individual has as smaller is the mutation probability.

For binary individuals mutation means flipping of variable values. For every individual the variable value to change is chosen uniform at random with low probability $P_m \in [0{,}001;0{,}01]$ (Fig.13.5).

In some cases *inversion mutation operator* is used. Two bits are chosen in individual at random and then chosen bits are exchanged (Fig.13.6).

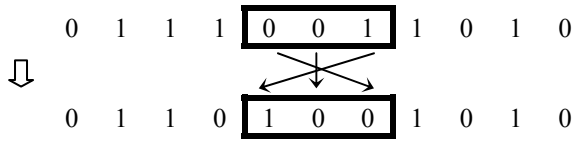


Fig.13.6 Inversion mutation

Note that mutation serves the crucial role for providing the gene values that were not present in the current population. It enables new individual properties acquisition. Thus mutation makes the entire search space reachable, despite population finiteness. In spite of the fact that crossover has the most efficient search mechanism, it does not guarantee the reachability for each point of search space.

So for solving any problem with genetic algorithm we must first of all define: *individual* and *population*, *genetic operators*, *fitness function*.

In ATPG problem solutions are represented as binary patterns or sequences of patterns also. Therefore it looks very attractive to use GA techniques for a decision of ATPG problems for DS at structural or functional description levels [2,3]. Further we will use different variants implementation and generalization of Gas for test generation problem of digital circuits.

13.2 Genetic test generation algorithm for digital circuits

The objective of digital circuits testing is to generate compact sequence of binary test vectors that has high coverage of manufacturing defects. The test sequence applied should be able to uncover all possible defects that could occur in manufacturing process. That is, the output response of defective chip (or board) should be different from the outputs of a good chip. At the same time real physical defects are modeled with faults, such as stuck at fault, short, bridge fault, transistor stuck open, transistor stuck close and so on. Mostly stuck at fault modeling is used in digital testing. Here nodes are assumed to be stuck at constant either '0' or '1' for the purpose fault modeling. So each node may have two types of this fault – *s-a*-0 and *s-a*-1. The approach usually used is to try to generate a test sequence that detects all single stuck-at faults in circuit under test. We would like to ensure, that generated test sequence contains a test for each single stuck-at fault in circuit under test. After a high fault coverage for single stuck-at faults is achieved,

the additional test sequences may be generated that target other fault models, such as transistor stuck, delay fault model so on.

Generally the test generation process consists of two phases. At the first phase there are used the methods that do not direct to specific single fault but take the all class of single stuck-at faults in circuit under test. Here test sequence is generated for faults that are checked enough easily with using not great computer power. Before the pseudorandom methods were used at this phase but now also genetic algorithms are exploited. Then the fault simulation determines the fault coverage and unchecked fault list for which it is necessary to generate test sequences.

So at second phase for each unchecked fault test sequence is generated with using deterministic (or genetic) algorithm. Then again the fault simulation is used for reducing unchecked fault list. This process cycles until the high fault coverage is reached. We will discuss the genetic algorithm using basically at the second phase of test generation. Where test sequence is generated for specific unchecked fault.

13.2.1 Test generation genetic algorithms for combinational circuits

At first the genetic algorithms were used for test generation of combinational circuits where output signals depend on only input signals and do not depend on state variables are usually represented with flip-flops. Here as a rule the individual corresponds to single binary test vector $X=(x_1,x_2,\ldots, x_n)$. The test generation problem may be formulated analytically for given single fault in one output combinational circuit[4]. Let $f(X)$ is a Boolean function implemented with good combinational circuit and $\varphi(X)$ - Boolean function implemented with fault circuit. Then Boolean expression $D(X)=f(X)\oplus\varphi(X)$ is called difference function. It is obvious that $D(X)=f(X)\oplus\varphi(X)=1$ defines the values of test vector X. So test generation problem is reduced to a solution search of Boolean equation $D(X)=1$. In the case multi outputs combinational circuit the difference function may be generalized in the following way:

$$D(X) = (f_1(X) \oplus \varphi_1(X)) \vee (f_2(X) \oplus \varphi_2(X)) \vee \ldots \vee (f_m(X) \oplus \varphi_m(X))$$

Also it is obvious that a solution of Boolean equation $D(X)=1$ gives the test vector for given fault.

This problem may be efficiently solved with using genetic algorithm. In this case the individual represents the binary test vector $X=(x_1,x_2,\ldots, x_n,)$ where $x_i=0,1$ and n equals circuit inputs number. So the population is composed of binary test vectors and standard genetic operators of crossover and mutation may be used in this case. Usually the number individuals in population is proportional to inputs number n (for example 3n)[5]. We will consider the genetic algorithm using for test generation for example of circuit fig.13.7.

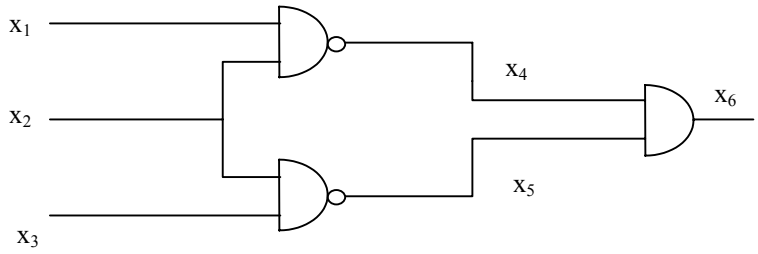


Fig.13.7 Combinational circuit

For the convenience there are represented in table13.1 all signal values for any inputs vectors.

Table 13.1.

X_1	X_2	X_3	X_4	X_5	X_6
0	0	0	1	1	1
0	0	1	1	1	1
0	1	0	1	1	1
0	1	1	1	0	0
1	0	0	1	1	1
1	0	1	1	0	0
1	1	0	0	1	0
1	1	1	0	1	0

Table 13.2

Input vector $(x_1, x_2, x_3,)$	Detected faults	Fitness function value $h= F_n *r$
000	$x_4 = 0$, $x_5 = 0$, $x_6 = 0$	3∗10
011	$x_2 = 0$, $x_3 = 0$, $x_5 = 1$, $x_6 = 1$	4∗10
101	$x_2 = 1$, $x_4 = 0$, $x_5 = 0$, $x_6 = 0$	4∗10
111	$x_6 = 1$	1∗10

From the beginning the initial test vectors population is generated in a random way. For example the initial population is shown in table 13.2 for circuit fig.13.7.

Here in second column the detected single stuck-at faults are shown for each population individual – binary test vector. Obviously that test vector detecting more faults should have more chance to be inserted in test sequence. So at initial stage we take the fitness function $h=F_n*r$, where F_n is number of newly detected faults with corresponding individual (test vector) and r is "bonus" for each detected fault (for our example r=10). In real program system number of newly detected faults F_d is determined with using of fault simulation. Obviously the best individual (binary vector 011 or 101) with maximum fitness function value $h=F_d*r$ must be inserted in test. Let for example the binary vector (101) is inserted in test. Further for next population generation it is necessary to apply the genetic operators – one(two)-point crossover and mutation. Once two individuals are selected, the crossover operator is used to generate two offspring.

As new individuals are generated, each bit is mutated with given small probability P_m. In simplest case of binary-coded GA mutation may be done by flipping a random selected bit. While in a non-binary coded, mutation involves randomly generated a new value in a specified position. So mutation produces incremental random changes in the offspring generated through crossover and brings new properties for individuals. Each new individual (test vector) must be evaluated with fitness function. Obviously the fitness function must take into account newly detected faults number for given test vector. Let after two generations current test set consists of two test vectors (101, 011), which detect (non detect) the faults shown in table 13.3. The current population of vectors is presented in table 13.4.

Table 13.3

Test vectors	Detected faults with current test sequence	Undetected faults with current test sequence
101	$X_2\equiv1$, $x_4\equiv0$, $x_5\equiv0$, $x_6\equiv0$	$X_1\equiv0$, $x_1\equiv1$, $x_2\equiv0$, $x_3\equiv0$, $x_3\equiv1$, $x_4\equiv0$, $x_4\equiv1$, $x_5\equiv1$, $x_6\equiv1$
011	$X_2\equiv0$, $x_3\equiv0$,$x_5\equiv1$, $x_6\equiv1$	$X_1\equiv0$, $x_1\equiv1$, $x_3\equiv1$, $x_4\equiv1$

Table 13.4

Input vector (x_1, x_2, x_3,)	Detected faults	Fitness function value h= F_d *s+ F_n *r
000	$x_4\equiv0$, $x_5\equiv0$, $x_6\equiv0$	0*10+3*1=3
001	$x_2\equiv1$, $x_4\equiv0$, $x_5\equiv0$, $x_6\equiv0$	0*10+4*1=4
100	$x_2\equiv1$, $x_4\equiv0$, $x_5\equiv0$, $x_6\equiv0$	0*10+4*1=4
110	$x_1\equiv0$, $x_2\equiv0$, $x_4\equiv1$, $x_6\equiv1$	2*10+2∗1=22

Note that here we have fitness function $h= F_d*s+ F_n*r$ where F_n is a number of newly detected faults and F_d is a number of earlier detected faults with corresponding individual (test vector). Here r=10 is bonus each newly detected fault and s=1 is bonus each early detected fault. In concordance with table 13.4 data the test vector (110) must be inserted in test sequence because it has maximum fitness function value. In the next table 13.5 the situation is shown for current test that consists of three vectors and has only two undetected faults.

Table 13.5

Test vectors	Detected faults	Undetected faults with current test sequence
101	$X_2\equiv1$, $x_4\equiv0$, $x_5\equiv0$, $x_6\equiv0$	$X_1\equiv0$, $x_1\equiv1$, $x_2\equiv0$, $x_3\equiv0$, $x_3\equiv1$, $x_4\equiv0$, $x_4\equiv1$, $x_5\equiv1$, $x_6\equiv1$
011	$X_2\equiv0$, $x_3\equiv0$,$x_5\equiv1$, $x_6\equiv1$	$X_1\equiv0$, $x_1\equiv1$, $x_3\equiv1$, $x_4\equiv1$
110	$X_1\equiv0$, $x_4\equiv1$	$x_1\equiv1$, $x_3\equiv1$

Similarly may be shown that at the next step the test vector (010) must be inserted in test, because it detects the last faults $x_1\equiv1$, $x_3\equiv1$. Overall test generation

genetic algorithm on base described approach of may be represented by way of following pseudocode.

```
Test generation(circuit)
      {
       Circuit initialization;
      Initial vectors population generation;
      While(stopping criteria met)
          {
           fault simulation;
           fitness function evaluation;
            insertion the best vector  in test;
          genetic operators execution;
          reproduction;
          crossover;
          mutation;
         new population generation;
          }
       test sequence output;
      }
```

Here at initialization stage the fault list is generated and other auxiliary operation are executed. Usually the initial vectors population is generated in a random way, but a priori and available information about good vectors may be used also. Fitness function evaluation is based on fault simulation. Note that in described approach the genetic algorithm solves at each step the local problem of next test vector search (not whole test sequence) in contrast basic GA, described in part 13.1 which is used as a rule for global problem solution. In next section we shall consider the global GA application for test generation for sequential circuits where the individual represent the whole test sequence but not single test vector.

13.2.2 Test generation genetic algorithms for sequential circuits

The test generation problem for sequential circuits is much more complex and its target setting depends on observation time test strategy [4]. Let good sequential circuit realizes finite state machine (FSM) $A=(X,Y,Z,\delta,\lambda)$, where X is the input set, Y is the set of states, Z is the output set, $\delta: Y\times X\rightarrow Y$ is the next state function, $\lambda: Y\times X\rightarrow Z$ is the output function. Since we consider the structure model of sequential circuit then functions δ and λ are implemented with combinational circuits accordingly Hafmen model

$$Y=(y_1,...,y_k),\ \text{where}\ y_i=(0,1)\ \text{for}\ i=\overline{1,k}; \qquad (13.3)$$

$X=(x_1,\ldots, x_n)$, where $x_j=(0,1)$ for $j=\overline{1,n}$; (13.4)

$Z=(z_1,\ldots, z_m)$, where $z_j=(0,1)$ for $j=\overline{1,m}$. (13.5)

Further we use the following notations[6]: $X(1), X(2),\ldots, X(p)$ denotes an input sequence of length p; $Y(y_0,0), Y(y_0,1),\ldots, Y(y_0,p)$ denotes the state sequence defined by initial state y_0; $Z(y_0,1),\ldots, Z(y_0,p)$ denotes the output sequence defined by initial state y_0 and input sequence $X(1), X(2),\ldots, X(p)$; $z_j(y_0,t)$ is the value at the j-th primary output after simulation step t. Using these notations the next state is defined by

$$Y(y_0,t)=\begin{cases} y_0 for = 0 \\ \delta(X(t),Y(y_0,t-1)) for t \neq 0 \end{cases}. \quad (13.6)$$

Similarly the output $Z(y_0,t)$ is defined by the function λ. A fault f transforms a state machine M into a machine $A^f=(Y,X,Z,\delta^f,\lambda^f)$, where functions δ^f,λ^f are defined analogically. Further we consider the different observation (respectively detection) time test strategy for sequential circuits.

Definition 13.1. A single stuck-at fault is detectable by input sequence X(1), X(2),…, X(p) with respect the single observation time test strategy (SOT) [6,7] if $\exists b\in\{0,1\},\exists t\le p,\exists i\le k,\forall(r,q):(z_i(r,t)=b\wedge z_i^f(q,t)=\overline{b})$, with r an initial state of fault-free circuit and q an initial state of faulty circuit.

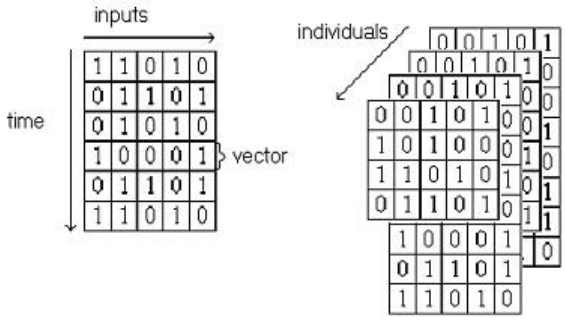
According above definition a fault is SOT-detectable if there is a unique moment t such that independent of the initial states r and q of good and faulty machines the output values on a particular output are different. For sequential circuits sometimes the other strategy is used that allows more precisely to define the fault detectability.

Definition 13.2. A single stuck-at fault is detectable by input sequence X(1), X(2),…, X(p) with respect the multiple observation time test strategy (MOT) [6,7] if

$\forall(r,q)\exists t\le p,\exists i\le k,\exists b\in\{0,1\}:(z_i(r,t)=b\wedge z_i^f(q,t)=\overline{b})$.

The fundamental difference between these two strategies is in following. According to MOT, the is an individual time moment for each possible initial state pair (r,q), such that output signals on particular output are different. Obviously that MOT strategy is more general the SOT. Some fault may be MOT-detectable but not SOT-detectable. So the test generation goal for sequential circuit is to find input sequence $X(1), X(2),\ldots, X(p)$ for that it holds true Def.13.1 or Def.13.2 depending on using strategy. It is natural that the second strategy requires more the computer resources. So we use basically the SOT strategy. But the genetic based

test generation algorithms allow to generalize the obtained results to MOT strategy in contrast structural methods where it is problematic.



a) individuals b) populations

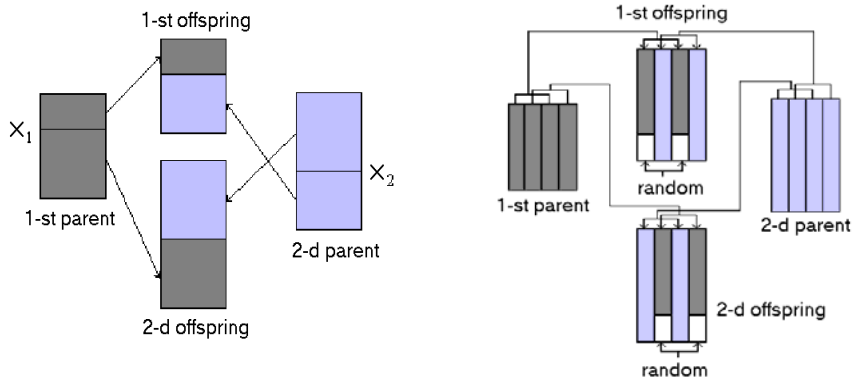
Fig.13.8 Encoding Of individuals and population in GA

Further for GA test generation of sequential circuits we will use as an individual a test sequence that is represented by binary table (Fig. 13.8 a). Here the column number is determined with circuit inputs number and the test length determines the row number. In this case the population consists of the fixed number of test sequences, possibly, different length (Fig.13.8 b).For the chosen encoding of individuals and populations the following problem oriented genetic operators can be used [4,6,8]:

Crossover.

1. The classic one point crossover. In this case the table is interpreted as one binary string.
2. The horizontal crossover where parents are crossed with subtables after some time point t<p as is shown at fig.13.9.
3. The vertical crossover where parents are crossed with random selected colomns as is shown at fig 13.9.
4. The free vertical crossover it is executed in the following way [keim]. Here crossover point is selected for each row and each pair is crossed by corresponding substrings. Note that this modification is the generalization of above vertical crossover.
5. The uniform crossover where each offspring gene is copied from one the parents accordingly random binary mask as it is shown at fig.13.4.
6. Structural crossover is the generalization of vertical crossover where also the parents are crossed by columns. Here it is used the exchange by columns groups corresponding to one treelike subcircuit. At that approach the exchange is directed to internal circuit check points that increases the test generation effectiveness for internal faults. Note that this crossover may be applied dynami-

cally that is the partition of circuit to treelike subcircuits is doing for specific fault of the given circuit.



a) horizontal crossover b) vertical crossover

Fig.13.9 Operations of the horizontal and vertical crossing GA

So crossover is implemented with using above six independent operators that are selected randomly with probability $P_1, P_2,\ldots, P_6$, which are derived experimentally under condition $P_1+ P_2+ P_3+ P_4+ P_5+P_6=1$.

Then as usually the generated offspring are mutated and three types of this operator accordingly are used with probabilities P_{m_1}, P_{m_2} and P_{m_3}:

1. Delete of one input vector from the by random chosen position. Application of this operation allows to reduce the length of the generated test sequence in that case, when a remote vector does not worsen test properties of sequence;
2. Addition of one input vector in random position, that also allows to extend the search area of decisions;
3. Random replacement of bits in a test sequence.

Similarly the random selection is used between these operators.

13.2.3 Problem-oriented fitness functions for test generation

The fitness function type plays key role in the GA-based search process. Therefore it is important to consider different types of fitness and evaluation functions, which are used in GA-based test generation methods.

The goal of testing process is to obtain different output values of good and faulty devices. Therefore the fitness function may be defined as measure of signal value changes in the simplest case [4]. In this case a fault free logical simulation may

be used. Another and more accurate approach is to define fitness function as measure of detected faults. In this case more complex fault simulation is used, but such approach allows obtaining quite good results. Obviously, that the number of signal value changes and the number of detected faults are important parameters having influence on the effectiveness of test generation process. There are certain parameters, which are important to the evaluation function definition and to the effectiveness of test generation for sequential circuits in modern test generation systems:

1. N – the number of nodes in circuit
2. N_d – the number of nodes with different values in the fault free and in faulty circuits
3. T – the total number of flip–flops
4. T_d – the number of flip–flops that changed state
5. E – the number of events in the fault free and in faulty circuits
6. L – the length of test sequence
7. F – the total fault number;
8. F_d – the number of detected faults;
9. F_{dt} – the number of faults propagated to flip–flops;
10. D – fault detectability;
11. W – sequence power;
12. O – flip-flop observability;
13. E_f – the number of events in the faulty circuit;
14. T_s – the number of flip–flops which are hardly to set.

In addition to mentioned above parameters the effectiveness of test generation algorithms depends on basic components of genetic algorithms - population size, crossover and mutation probability, generation numbers etc.

In CRIS [9] the hierarchical simulation technique is used that allows to reduce memory expenses and to deal with very large circuits. The classical GA is used, in which population evolves from generation to generation through reproduction, crossover and mutation. Each individual represents the test sequence. System CRIS is based on continued mutation of test sequence and its analysis with simulation procedure. Given system demonstrates good results (it is fast and produces compact test sequences with high fault coverage for combinational and sequential circuits) but has essential drawback – the manual tuning GA parameters for each circuit.

System GATEST [10,11] is oriented to the sequential digital circuits and based on two–level GA. The first level GA generates single test vectors; the second level GA generates test sequences from these obtained vectors. Accordingly in the first level GA individuals correspond to single test vectors, while in the second level GA they are test sequences. GA uses different crossovers:1–pointed, 2–pointed, uniform.

The first–level GA is subdivided to three phases. Thus GA has four phases. The evaluation functions are different according to the algorithm phase:

- in the first phase, the algorithm goal is the flip–flop initialization and so evaluation function is defined as follows

$$h_1 = T + \frac{T_d}{T}, \tag{13.7}$$

here evaluation process uses only the fault-free logical simulation;

- in the second phase, all flip–flops are assumed to be initialized, and the goal is finding new test vectors able to detect additional faults; so evaluation function is

$$h_2 = F_d + \frac{T_d}{FT}; \tag{13.8}$$

- the third phase comes if the generated vector does not detect additional faults and uses the following evaluation function

$$h_3 = F_d + \frac{F_{dt}}{FT} + \frac{E}{NF}; \tag{13.9}$$

if the generated vector detects additional faults than algorithm returns back to phase 2; otherwise if number of the unused vectors exceeds the definite limit than algorithm comes to phase 4;

- in the fourth phase test sequences are generated from designed vectors and GA uses the following evaluation function

$$h_4 = F_d + \frac{F_d}{F\,TL}. \tag{13.10}$$

In phases 2–4 evaluation uses the fault simulator that slows down the test generation process. This package show good results: high fault coverage and low execution times for sequential benchmark circuits. But it also has the same drawback – the manual tuning GA parameters (alphabet size, iterations number, population size, mutation rate).

The interesting approach is used in package DIGATE [11,12]. It is organized in two phases:
– the first phase selects a target fault and GA activates it to flip–flop;
– in the second phase GA searches the sequence that makes the target fault observable at the circuit primary outputs. It uses the distinguishing sequences that

able to propagate a fault effect from flip–flop to primary outputs. The distinguishing sequences are pre–computed and stored for future use. The test sequences are constructed as concatenation of activating and distinguishing sequences. Evidently here each individual is a sequence. The evaluation uses the fault simulation. Accordingly the phase 1 and 2 have the following evaluation functions

$$h_1 = 0.2D + 0.7(W + O) + 0.1(E_f + T_s + T_d), \tag{13.11}$$

$$h_2 = 0.8D + 0.1(W + O) + 0.1(E_f + T_s + T_d). \tag{13.12}$$

The weighted coefficients were found heuristically for each phase, but they are universal for any circuit. It is advantageous difference of considering method from previous packages.

In another system GATTO the individuals are input sequences too [13]. In this package the basic effort is directed to determination of the evaluation function as the measure of closeness to the optimum solution. The individuals are evaluated with fault simulation according to their activity (the more lines with different signal values in good and fault circuits the more value of detectability probability). So the evaluation function depends on three basic parameters: the weighted number of gates with different signals in good and faulty circuits, the weighted flip–flop number with different signals in good and faulty circuits, and the sequence length. The weights are empirical measures of gate and flip–flop observability accordingly. The last parameter is used for improvement of test sequence compactness. So evaluation function for a single input vector combines the above parameters

$$h(v, f) = c_1 f_1(v, f) + c_2 f_2(v, f), \tag{13.13}$$

here f is the fault being considered and v is input vector; c_1 and c_2 are normalization constants, while f_1 and f_2 represent the weighted sums mentioned above.

The evaluation function H for the entire sequence s is computed according to the best vector it contains

$$H(s, f) = \max_{v_i \in s}(LH^i * h(v_i, f)). \tag{13.14}$$

Here constants $LH \in (0;1)$; i is a position of the vector v_i in the sequence s. Due using this evaluation function shorter sequences are preferred and the final test length is reduced.

This approach does not contain the handle tuning of GA parameters for each circuit also.

The most effective is the test generation method that combines advantages structural deterministic and GA-based approaches. In this case the phase of entering fault and primary output activation is executed with deterministic method and multivalued logic, and the phase of justification – the search of circuit inputs justified requirements obtained at first phase, is carried out with GA. Also the perspective approach is application of parallel GA where several populations of solutions are simultaneously evolved, basically independently of each other. But time to time the exchange of the best solutions is executed between populations in different methods. The nature of test generation problem is hierarchical therefore it is reasonably to use hierarchical GA, where at every level different GA is applied.

13.2.4 GA test genetation implementation

The general approach to genetic test generation lies as follows. We use the individual encoding with binary tables as shown above at previous section in fig.13.8. For such kind of individual encoding and population representation the above described special problem-oriented genetic operators are used. The test generation process contains three phases. The first phase goal is the fault sensitization. Here the signal difference good and fault circuit is propagated to pseudooutputs. Then the second phase is executed for test properties improvement for sequence generated at first phase. This phase is algorithm kernel and use the genetic algorithm. After that at second phase the test sequence is generated the fault simulation is necessary to determine the undetectable faults.

The general pseudocode of test pattern generation is bellow.

```
Test_ generation(sequential circuit)
{

 fault set generation();
 while(fault coverage < given threshold)
 {
  //Phase1
  goal = fault sensitization();
  if  (goal == empty set)
      exit;
  //Phase2
  sequence = GA test sequence generation(goal);
  // Phase3
  if (sequence != empty)
     fault simulation(sequence)
```

```
    else          // test sequence for goal fault not found
        mark fault as untested();
  }
 }

GA test sequence generation(goal=f)
  {
   For ( i=0 ; i<MAX_GENERATIONS; i++)
   {
    For ( each individual s in population P )
     Fitness_evaluation(s, f );
    new  P=∅;
    for  (k=0 ; k< MAX_NEW_INDIVIDUALS  ; k++ )
    {
     selection of 2 sequences s1 and s2 in P;
     crossover(s1,s2); //генерируются две особи
     mutation(si);
     newP=newP∪s;
    }
    P=(best MAX_individuals from newP and P );
    for( each individual s in population P)
     if( s detect f )
      return s;
   }
   return( no_sequence )
 }
```

Two fault simulation algorithms were integrated in order to accelerate test generation. The first one is single pattern parallel fault propagation (SPPFP) method that is used in phase 1 for checking activation of any given fault by randomly generated sequence. Here the fitness functions defined with formulas (13.13, 13.14) are used. The evaluation function is computed with the help of second fault simulation algorithm that belongs to the group of parallel pattern single fault propagation (PPSFP) methods. The second one was developed especially to using in GA based test generation algorithm.

13.3 Distributed test generation methods

Today numerous modifications and generalizations of GA are suggested [69]. The parallel GA (PGA) are roughly upcoming and very promising from the theoretical investigation and practical application points of view [14-16].

13.3.1 GA parallelization

Inherent GA "internal" parallelism and possibility of the distributed calculations promote to development of parallel GA. The first papers in this direction were appeared in 60-th years. But only in 80-th years, when accessible tools of parallel realization were developed, the PGA investigations adopted systematic mass character and practical orientation. Numerous models and realizations were developed in this direction, some of which are represented below .

First of all necessary to note that the basis of PGA is population structuring – decomposition to few subpopulations (subsets). This decomposition can be fulfilled with different methods, which define different types of PGA. Further we shall consider the modern main methods of the PGA realization.

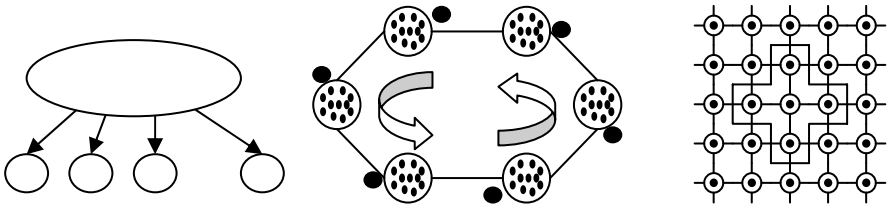


Fig.13.10 Different types of parallel GA: a) global PGA, b) distributed PGA, c) cellular PGA.

Most known is global parallelism which represented on Fig. 13.10.a). This model is based on simple (classic) GA in which the fitness function calculations are performed in parallel. This approach is faster, than classic sequential GA and does not usually require balance on the load as on different processors. This model often named "**master-slave**". Many researchers use the pool of processors for the increase of speed execution of algorithm. At the same time the independent program running of algorithm at different processors are executed essentially faster than at one processor. It must be noted, that in this case there is no co-operation between different runs of algorithm. It is extraordinarily simple method of parallelization and it can be very useful. For example, it can be used for the decision of

the same task with different initial conditions. GA allow effectively use this method by virtue of their probabilistic nature. At the same time we have minimum program changes, but advantages are considerable.

In *distributed PGA* (Fig.13.10.b) a population is divided to a set of subpopulations, which evolve independently (accordingly to simple GA) and can communicate with neighbor subpopulations in certain manner after some “isolation time”. This parallel paradigm is often implemented in an extraordinarily popular "*model of islands*" (coarse grain), where great number of subalgorithms simultaneously work in parallel, exchanging in the search process by some individuals. This model assumes direct realization on the computing systems with MIMD- architecture. Thus every “island” corresponds to its own processor.

In *cellular PGA* (fine grain) (Fig. 13.10.c) there is a set of subpopulations consisting of only one individual. Given individual-subpopulation can communicate only with neighbor individuals-subpopulations at once. A neighbor relationship is defined as certain regular structure –grid (Fig.13.10.c). For cellular PGA parallelism is usually implemented on the computer systems with SIMD-architecture, where every processor represents subpopulation-individual. Although another papers are known where authors use single possessor computers and systems with MIMD-architecture.

13.3.2 Parallel test generation method based on the “master-slave” model.

In this section for parallelization of GA we use a model «master-slave», because it requires the small changes in the existing software implementation of test generation GA and gives quite good results.

In this approach every processor has its own copy of population. The calculation expenses of fitness-functions values (witch use a logical simulation) are evenly distributed to all processors. For all processors the same list of faults is used. Therefore for n individuals and P processors we take the P/n individuals to every processor. The values of fitness-functions are calculated by the slave processors and are sent to one selected processor-master, which collects all information and passes it to all processors. Every processor has information about the values of fitness-function for all individuals and can create next population generation on this basis.

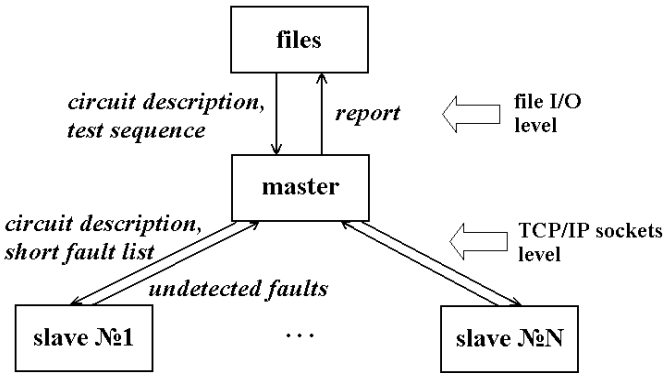
So the processor-master executes central part (kernel) of test generation algorithm, while the logical simulation (fault-free and fault) of digital circuits are implemented on processors-slaves. The fault simulation is most critical with point of view of calculation expenses. Different methods of the distributed fault simulation are known, which are mainly based on decomposition: 1) circuits on subcircuits; 2) test sequence on subsequences. We will take combined approach of these two methods.

On the first and second stages the simulated input sequences are distributed between working processors. On the first stage every working processor is loaded by the generation (simulation) of one subsequence. For balance the list of undetected faults is broken up on approximately identical subgroups.

At the end of each of three stages the points of synchronization are placed. When a processor-master arrives at these points, it goes to the wait mode, while all working processors will not make off the tasks that guarantee global correctness of algorithm. Thus work is distributed between a processor-muster and workers as follows.

Processor-master:

- Performs all input-output operations with an user and file system: it reads circuit description and fault list, and writes the generated input test sequence;
- Initially runs «slave» processes on available resources;



- Distributes the copies (internal form) of circuits and fault lists to every working processor;
- Organizes the process control of test generation: as soon as input sequence has to be fault simulated, it sends the proper message for activating of working processors; when working processors finish their work, processor- master receives results and accordingly changes global data structures (general fault list, values of fitness-functions for individuals etc).

Fig.13.11 Data flow diagram for distributed test generation and fault simulation algorithms

A processor-worker keeps the local copy of circuit (in internal format) and fault list. Every «worker» takes an input sequence from the «master» and determines the faults are detected by this sequence by the logical simulation and calculates the values of fitness-function for individuals. It sends the obtained results to the master and wait next task. As the population size is much larger than a number of processors, good balance in the load of processors is achieved. For every working

processor the change of local fault list with the detected and undetected faults from other working processors requires a lot of resources and it is critical.

Final results (test input sequences and fault coverage) are near to the results obtained on the single possessor computer system with the use of a similar algorithm. Quality of decision (test fault coverage) is not here lost and is in most cases got better, and time of test generation grows short substantially. The data flow diagram of considered algorithm is cited on fig.13.11.

13.3.3 Distributed fault simulation

Described above distributed genetic algorithm of test generation is based mainly on the distributed fault simulation algorithm. Now we will shortly describe also this method.

One of the central problems of digital device diagnostic is fault simulation of digital circuits. And persistent increasing of modern device complexity makes the task of reducing fault simulation time still actual. One of possible ways to speed up fault simulation procedure is adaptation of existing algorithms for multiprocessors computing systems (clusters) implementation.

Distributed fault simulation is organized in similar way and is based also on the «master-slave» approach like distributed test generation. One processor here is selected as master and remained processors – as slaves. There exist several approaches to implementation of distributed fault simulation: partitioning of circuit and partitioning of fault list . Our algorithm is based on the fault list partitioning.

Data flow chart for this scheme of computational process is showed on fig.13.11.

Every slave processor performs fault simulation on the data received from the master: circuit description and fault sublist. The pseudocode of this process is given below.

```
slave_process_fault_simulation()
{
 search_of_master_process();
 if( master_was_found )
 {
  receive_circuit_description();
  receive_fault_sublist();
  parallel_fault_simulation()
  send_list_of_undetected_faults();
 }
}
```

The kernel of this process is the procedure *«parallel_fault_simulation»*, which is a regular fault simulator that used in single processor implementation. In our case we used home built PROOFS-based fault simulator, described in [17]. Mark the main advantages of this algorithm that makes it very successful: 1) dynamic fault-list processing: detected fault is eliminated from fault list in the same time it detected, no simulation performs for this fault further; 2) fault sorting which allows include in one group the faults that cause the same simulation events; 3) the technique of functional fault injection.

Common data flow chart diagram that describes interaction among master and slaves processes is shown on Fig.13.11.

It is necessary to notice that master process performs two types of exchange operation. File input/output operations are necessary to obtain circuit description and test sequence to be simulated. In contrast all data interchange among master and slave process is performed via TCP/IP sockets. This fact enables to construct computing cluster on the common used computers. Authors used as such cluster 100Mbit local intranet.

Data flowchart diagram shows that master processor does not perform any simulation but organizes the computing:

- Reads the circuit description to be simulated and input test sequence;
- Sends this description and test sequence for all client processors;
- Receives from slaves fault simulation results and makes common report.

Algorithm for master process for distributed simulation is given below.

```
distributed_simulation(circuit,test)
{
 number_of_slaves = search_of_slaves();
 if( number_of_slaves != 0 )
 {
  input_circuit_description();
  input_test();
  make_full_fault_list();
  partiting_the_fault_list(number_of_slaves);
  for( i=0 ; i< number_of_slaves ; i++ )
  {
   send_to_client_i_circuit_description ();
   send_to_client_i_part_of_fault_list();
   send_to_client_i_test_sequence();
  }
  for( i=0 ; i< number_of_slaves ; i++ )
  {
   receive_list_of_undetected_faults();
  }
  make_report();
```

```
    }
}
```

Master process starts with search procedure of calculation clients. Further it divides full fault list into sublists prorate number of found clients. Then for all clients the following operation sequence is fulfilled in cycle: sends circuit description in internal format; sends test sequence and corresponding short fault list. After that master transfer to state of waiting data from clients. At the next step master receives the results of fault simulation from each client and makes general reports: fault coverage, common simulation time, time of simulation on every clients. The constructed in described way distributed fault simulation algorithm allows a high parallelization of simulation process.

13.3.4 Distributed test generation based on the "model of islands"

In this section for GA parallelization the "model of islands" is used. Here separate subpopulation, which is initialized randomly and evolved independently, is realized on each processor. In given iteration number subpopulations are exchanged by some individuals in certain way. Each processor select the best individuals of own subpopulation and send them to neighbor processors subpopulations (neighborhood concept is a parameter of method). These individuals are accepted in neighbor processors subpopulations and then independent evolution on each processor-“island” is continued.

In this approach there are more chances to obtained high-quality solution, since different areas of search space are investigated on different processors [90-92]. Moreover in this case it is possible the reducing of search time due to the best individual migration.

In contrast to previous method (“master-slave” model), where GA works only on the central processor-master and processors-slaves are used only for fitness function computing, in this approach full GA is implemented on every processor. In other words each processor executes full cycle of GA evolution operations: fault-free and fault logical simulation, test sequences generation. Each processor works with full circuit and fault list. At the same time there are two reasons of speeding-up test generation process at least. Every processor operates with subpopulation of less dimension that less time is required. Due to the best test sequences migration each processor can detect faults quicker then in case of independent operation in subpopulations. One of the most important parameter of this model is population power (individual number) of subpopulation. The influence and selection of this parameter will be considered below.

The main factors that affect on migration in "model of islands" (hence it affect to effectiveness) are as following:
Migration rate – a number of exchanged individuals;
Selection method of individuals for migration;

Isolation time which defines generation number between migration phases;
Strategy of individual replacement with migrated individuals from neighbor subpopulations; here also there exist different approaches the worst individuals are replaced, random individuals are replaced in subpopulation etc.;
Replication strategy of migrating individuals. Under first approach migrating individual also stays in starting subpopulation. The second approach demands removal of migrating individual from starting subpopulation. The first strategy can result in domination the same strong individuals in different subpopulations. Under the second strategy an individual can return back to start subpopulation in some time that results in extra computing expenses;
Topology which defines neighborhood relationship between subpopulations, here exchange is fulfilled only between neighbor subpopulations.

There exist several standard methods of selected individuals exchange between subpopulations. Time expenses to individual migration between subpopulation depend of used exchange method.

- Exchange by the ring:

In this method individuals can migrate to one neighbor subpopulation. In this case the number of $m = n - 1$, where n – the number of computers;

- Two-way exchange by the ring:

Here, likewise to previous method, exchange of individuals is executed between the closest neighbors, and neighborhood relation is defined by two-dimensional structure.

13.3.5 Implementation and experimental investigations of distributed genetic algorithms of test generation and simulation

Developed algorithms were implemented with using blocking sockets technology in C++ Builder programming environment. For computer experiments the computing cluster on the base of 100 Mbit local intranet was used. The cluster nodes have following parameters: Intel Celeron 2 GHz processor, RAM 256 Mb, OS Windows XP.

For research of effectiveness of suggested algorithms following time parameters were calculated during computer experiments: whole time of simulation process, events number in fault-free and fault simulation, whole number of events. For comparison the experimental results of algorithms from [2] were taken.

At first let consider experimental results obtained for distributed GA implementation based on the “worker-farmer” model. The diagram of simulation speeding up for circuit s35938 (ISCAS89), under condition of change of processors-client number from 1 to 8, is represented on fig.13.12. Given experimental results

confirm the effectiveness of suggested parallelization method of simulation algorithm. Finally on the fig.13.13 the simulation results for large circuits of ISCAS benchmark are represented. These data show the relative speeding-up with increase of circuit size. It is explained by that fact that for large circuits the expenses of parallelization are reduced compare with fault simulation expenses.

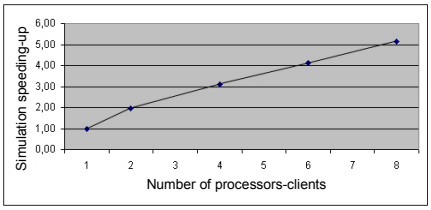


Fig.13.12. Speeding-up of fault simulation for circuit 35938 according to worker number

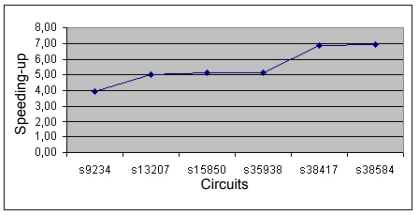


Fig.13.13. Speeding-up according to circuit complexity

Table 1

Circuit from ISCAS89 benchmark	Speeding-up relatively to one processor	Fault coverage increase
S1196	1.59	+0.8%
S1238	1.8	+0/6%
S1423	1.1	+12.8%
S1488	6.1	+7.1%
S5378	5.16	+1.3%
S35932	5.35	+1.6%

Further let consider the results of implementation of test generation distributed GA which is based on the “islands model”. In table1 there are represented experimental results, which show the speeding-up and test quality for several circuits from ISCAS89. In this case 8 processors and ring migration method were applied.

Obtained results confirm the effectiveness of test generation and fault simulation algorithms parallelization. The comparison of experimental results show that “farmer-worker” model gives more speeding-up in comparison with “island model” relatively to one-processor system and essentially easier in software implementation. But “island model” raises fault coverage of generated tests especially for large circuits. Therefore parallelization based on the “island model” has a reason only in case when generated tests have unsatisfactory fault coverage for “farmer-worker” model.

13.6 Hierarchical GA of test generation for highly sequential circuits

Usually the sequential circuits have the (re) set sequences that allow to install the memory elements to some determined states. In this case the test generation is essentially simpler. The general and hard case of testing sequential machines is testing sequential machines without possibility to set it in initial state. This kind of machines with memory is often called highly sequential machines or circuits. In this case other approaches are applied [11,18].

Hierarchical approach can be effectively applied and implemented for sequential circuits at structural (gate) level. In this case at low level the some characteristic sequences are generated and then are used under test generation at high level.

Given approach is applied to test generation in highly sequential circuits for hard-to-test faults . Here two-phase strategy is used for test generation: 1) fault activation; 2) fault propagation. The iterative combinational circuit is used as model of sequential circuit (Fig.13.14). At first phase an attempt is made to derive a sequence that activates the chosen fault and propagates its effect to primary outputs (POs) or flip-flops (FFs). At second phase the fault effect is propagated from FFs to POs of iterative combinational circuit with assistance of distinguishing sequences basically. So, the basic problem in fault activation is transition of circuit under test (good and faulty) to specified set conditioned by obtained FFs input values (pseudo inputs).

A transition sequence is generated with assistance of genetic algorithm. In order to generate such kind of sequences with assistance of the dynamic state traversal algorithm, a table of visited states is mapped to the list of input vectors in the test set [18]. However, if ending state was not visited then transition sequence is generated with help of GA. In this case initial population consists of random sequences of given length and the sequences, which solve problem partially. For example, it can be input sequences that set only some FFs to necessary values. In this case different input sequences can set different FFs to necessary values and, obviously, can be useful in transition sequences generation.

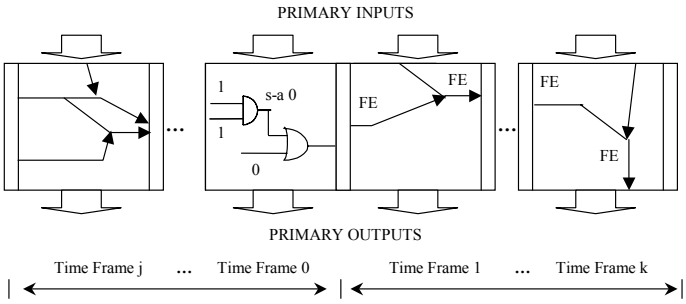


Fig.13.14 Two-phase strategy test generation in iterative combinational circuit

In this case at low level some characteristic sequences are generated that permit to set the flip-flops at some deterministic values that simplify a problem test generation for sequential circuits.

Characteristic sequences.

For test generation at high level there are useful the following input characteristic sequences. First of all we define set and reset sequences as follows:

1. **S_i-sequences.** A flip-flop **set sequence** is a sequence that sets the i-th flop to a 1-state;
2. **R_i-sequence.** Similarly a flip-flop **reset sequence** is a sequence that resets the i-th flip-flop to a 0-state.

These sequences associated with flip-flops are intended to (re)set the flip-flops starting from an unknown state. Such S_i (R_i) – sequences are called **type A** and generated at preprocessing step of test generation. The sequence type A length is restricted with 4D, where D is sequential depth of circuit. If S_i (R_i) – sequences require some flip-flops must be (re) set to specific (not arbitrary) states that these sequences are called **type B**. These sequences are generated dynamically in case of need during test generation process.

3. A **pseudoregister justification sequence** is a sequence that is able to justify (set or reset) the required flip-flops states for particular pseudoregister. Here the pseudoregister is the group of flip-flops.

At second test generation phase a distinguishing sequences propagating fault effect from FFs to Pos are required. In this case three types of the distinguishing sequences are used [18] (Fig.13.15).

4. The **distinguishing sequence of type A** for FF *i* is defined as a sequence that produce two distinct output responses when applied to the fault-free DD for two initial states, and initial states differ in the *i*-th position and are independent of all other FF values.
5. The **B-type distinguishing sequence** for FF *i* is a sequence that, when applied to the fault-free DD with *i*-th FF = 0 (or 1) and applied to the faulty DD with the same FF = 1 (or 0) for two initial states, produces two distinct output responses independent of all other FF values.
6. The **C-type distinguishing sequence** is similar to type B except that the subset of FFs are assigned to specific logic values.

For every distinguishing sequence the “distinguishing power” is assigned. It evaluates the possibility of given distinguishing sequence to propagate fault effect from according FF to PO. A distinguishing sequence has major “distinguishing power” if it is necessary to set specified values to small number of FFs. Also distinguishing sequences, which are able to propagate effects of several faults, have greater “distinguishing power”.

Type A		Type B		Type C	
Fault-free DD	Fault-free DD	Fault-free DD	Faulty DD	Fault-free DD	Faulty DD
FFs	FFs	FFs	FFs	FFs	FFs
Uuu1uuu	uuu0uuu	uuu1uuu	uuu0uuu	uuu1Suu	uuu1Suu

Fig.13.15 Types of distinguishing sequences

The described characteristic sequences are generated with using the low level genetic algorithm. In this GA individuals are represented with binary tables and the problem oriented genetic operators (crossover and mutation) adjust to these tables as shown in part 13.2.2. In this case initial population consists of random sequences of given length and the sequences, which solve problem partially. For example, it can be input sequences that set only some FFs to necessary values. But different input sequences can set different FFs to necessary values and, obviously, can be useful in transition sequences generation.

During test generation the high level genetic algorithm uses as fabricated parts the characteristic sequences which are generated at low level. It makes the evolutionary search more directional and effective. At the high level the modified genetic algorithm is used. In the first place the initial population includes not only random binary tables, but also the generated characteristic sequences. In the second place more extensive set of genetic operators is used during test generation.

Note that the different fitness functions are used at the various level and phases. Since fault activation and fault propagation phases target different goals, their corresponding fitness functions are differed. The used parameters are as follows:

- P_1 – fault detection;
- P_2 – sum of dynamic controllabilities;
- P_3 – matches of FFs values;
- P_4 – sum of distinguishing powers;
- P_5 – induced faulty circuit activity;
- P_6 – number of new visited states.

Parameter P_1 is self-explanatory, in particular during the fault propagation phase. It is included in fault activation phase to cover faults that are propagated directly to the POs in the time frame in which are excited. P_2 indicates the quality of states to be justified. P_3 guides the GA to match the required FFs values in the state to be justified during state justification, from the least controllable to the most controllable FF value. P_4 measures the quality of the set of FFs reached by the fault effects. P_5 evaluates the number of events generated in the faulty circuit, with events on more observable gates weighted more heavily. P_6 is used to expand the search space. Thus, on basis of considered parameters following types of fitness functions are used:

Fault activation phase:

- Multiple time frames

$$F_1=0,2P_1 + 0,8P_4; \tag{13.15}$$

- Single time frame

$$F_2=0,1P_1 + 0,5P_2 +0,2(P_4 + P_5 + P_6); \quad (13.16)$$

- state justification

$$F_3=0,1P_1 + 0,2(k - P_2) + 0,5P_3 + 0,2(P_5 + P_6), \quad (13.17)$$

where k is a constant;

Fault propagation phase:

$$F_4=0,8P_1 + 0,2(P_4 + P_5 + P_6). \quad (13.18)$$

Note that large value of weight coefficient of the P_4 is used in fitness function at the activation phase. If activation sequence for target fault cannot be generated directly then this problem is solved in few stages: at first the fault activation is fulfilled within a one iteration of combinational iterative circuits and then the sequence for setting flip-flops to target state is generated. Obviously, that the main parameter is the number of detected faults at the propagation phase. Therefore coefficient of P_1 has enough large value. Note that GA cannot find out undetectable (redundant) faults. Therefore it is desirable to use a deterministic test generation method for residuary undetectable faults.

13.7 Genetic programming in test generation of microprocessor systems

Testing of microprocessor-based systems is a very serious problem. The most complicated task is that of generation of test sequence. Traditional structural methods of test generation, which normally require the description of the logic circuit structure on the gate level, are not applicable for such systems owing to very high task dimension. The generation of test-programs of microprocessor systems (MS) usually was carried out at function level practically “manually”. At that the test represents an assembler-program unlike binary sequences for logic circuit.

One of the most perspective approaches to the MS test pattern generation is approach based on the genetic programming (GP). Checking sequence for MS is test program consists of assembler language operators. Classical GP uses for individual representation tree-like structures that does not allow operate arbitrary programs. Therefore in given case graph-based program representation, especially directed acyclic graph (DAG), is applied (Fig.13.16) [19].

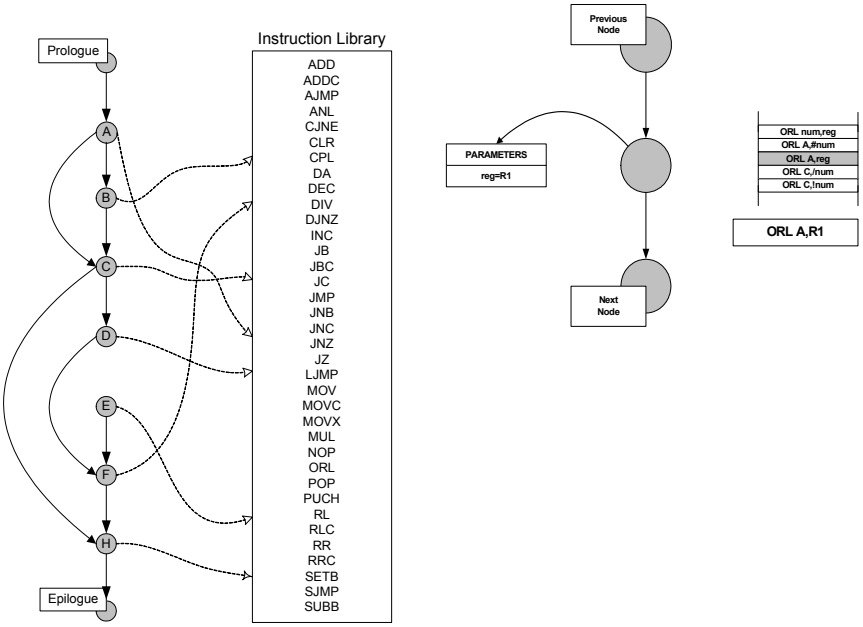


Fig.13.16 DAG and Instruction Library (on the left), a sequential instruction and its parameters (on the right)

Each node of the *DAG* (Fig.13.16) contains a pointer to the instruction library and, if necessary instruction parameters (i.e., immediate values or register specifications). The instruction library describes the assembly syntax, listing each possible instructions with the syntactically correct operands. Although instruction library may also contain macros instead of instruction, with the exception of prologue and epilogue, all entries correspond to individual assembly instruction. For instance, (Fig.13.16) shows a sequential node that will be translated into an "*ORL A, R1*", i.e., a bit-wise OR between accumulator and register R1. *DAGs* are built with four kinds of nodes:

- *Prologue* and *epilogue* nodes represent required operations, such as initializations. They depend both on the processor and on the operating environment, and they may be empty.
- *Sequential-instruction* nodes represent common operations, such as arithmetic or logic ones (e.g., node **B, F** (Fig.13.16)). *Unconditional* branches are considered as sequential, since execution flow does not split (e.g., node **D** (Fig.13.16)).
- *Conditional-branch* nodes are translated to assembly-level conditional-branch instructions (e.g., node **A** (Fig.13.16)). All common assembly languages implement some jump-if-condition mechanisms. All conditional branches implemented in the target assembly languages must be included in the library.

Test programs are induced by modifying *DAG* topology and by mutation parameters inside *DAG* nodes. Following genetic operators (mutation and crossover) are applied:

- ***Mutation 1 - Add node*:** a new node is inserted into the *DAG* in a random position. The new node can be either a sequential instruction or a conditional branch. In both cases, the instruction referred by the node is randomly chosen. If the inserted node is a branch, ether unconditional or conditional, one of the subsequent nodes is randomly chosen as the destination. Remarkable, when an unconditional branch is inserted, some nodes in the *DAG* may become unreachable.
- ***Mutation 2 - Remove node*:** an existing internal node (except prologue or epilogue) is removed from *DAG*. If the removed node was the target of one or more branch, parents' edges are updated.
- ***Mutation 3 - Modify node*:** all parameters of an existing internal node are randomly changed.
- **Crossover:** two different programs are mated to generate a new one. First, parents are analyzed to detect potential cutting points, i.e., vertices in the *DAG* that if removed create disjoint sub-graphs. Then a standard 1-point crossover is exploited to generate the offspring.

Fitness-function of the second level is build on the basis of coverage measure of VHDL operators. Thus fitness-function exploits the data obtained by means of Active VHDL (code coverage).

During construction of tests for microprocessor system is used the following fitness-function:

$$F = c_o N_{ao} / N_o + c_b N_{ab} / N_b + c_b N_{ac} / N_c \,, \tag{13.19}$$

where N_{ao} –the number of linear statements VHDL have been activated by test-program; N_{ab} –the number if statements have been activated by test program; N_{ac} – the number of case statements have been activated by test program, $N_{o,}$ N_b, N_c the common number of *linear*, *if*, *case* statements accordingly; c_o, c_b, c_c – normalizing constants ($c_o + c_b + c_c = 1$).

The program implementation (Fig.13.17) is carried out in the Active VHDL environment in accordance with the following scheme. The present population of test-program (in assembler) is being generated by the method based on the genetic programming which is implemented beyond Active VHDL.

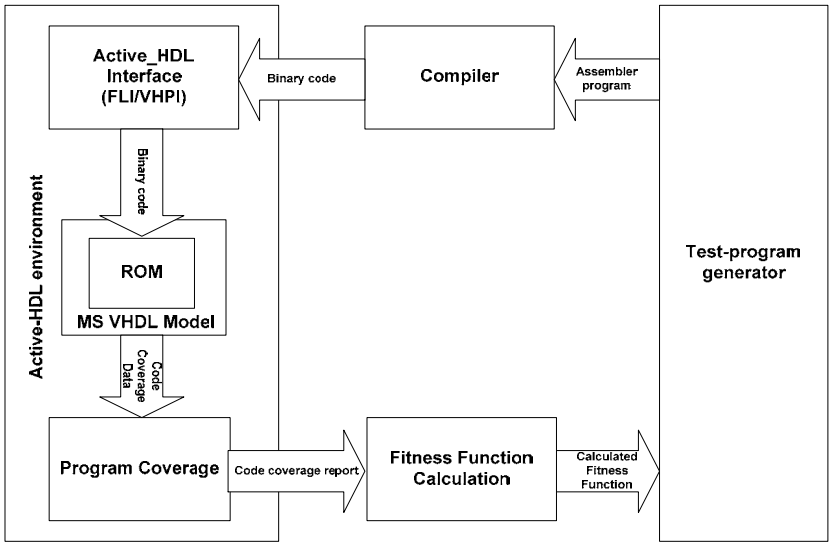


Fig .13.17 Program implementation

The algorithm of test program generation is presented below as pseudo code.

generation of test-program initial population;
While (not attained maximum number of generation)
// loop according to generations
Generation of various paths for each test program;
While (not attained stop condition)
//loop according to paths
{
Test-program generation according to correspondent path
Compilation of test-program to binary code
Entry to Active VHDL environment
Loading of binary code to ROM of microprocessor
system VHDL model
Estimation of test program coverage using Active VHDL
Exit from Active VHDL environment;
Calculation of fitness-function according to correspondent path;
} // end of loop according to paths

Calculation of fitness function for test-program
(graph);
//creation of the next generation;
Selection of parents according to fitness-function value;
Crossover;
Mutation;

Reduction of population;
} //end of loop according to generations

Generation of initial population is implemented on base graph of test program representation by link list of neighbor nodes. For each node of graph the correspondent link list of adjacent nodes have been processed. The graph complexity may be vary by tuning the following parameters: number of nodes in graph, number of macros and number of successor-nodes for each node.

The approbation of the presented approach is done for microcontroller 8051, the model of which is given on the function level in the VHDL language. The analyses and comparison of simulation data of circuits at the function and logic level show that generated test-programs have high fault coverage. At the same time the generation of checking tests on function level is being done essentially faster.In Fig .13.18 the results of genetic algorithm of test program generation's implementation for microcontroller 8051 are shown graphically. As we can see the fitness function has reached its maximum value equal 97.36 %.

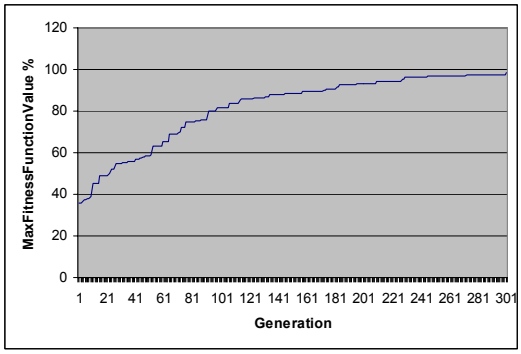


Fig .13.18 Fault coverage

Conclusions

This paper presents new perspective approach to DS test generation that is based on using evolutionary algorithms and hierarchical solution. It was shown that given approach could be effectively applied to test generation at basic DS representation levels:FSM and structural, for highly sequential circuits, 2-levels hierarchical genetic algorithm is applicable;MS level – GP-based approach is applied.

References.

1. Holland J.P. Adaptation in Natural and Artificial Systems.An Introductionary Analysis With Application to Biology? Control and Artificial Intelligence.Univercity of Michigan,1975.-210 p.
2. Goldberg D.E. Genetic Algorithms in Search, Optimization and Machine Learning.-Addison-Wesley, reading,MA.-1989.
3. Skobtsov Y.A.,Skobtsov V.Y. Modern modifications and generalizations for genetic algorithms//Tavrichesky bulletin of information science and mathematics. – 2004. – №1. – C.60-71 (in Russian).
4. Y.A.Skobtsov, V.Y.Skobtsov. The logical simulation and testing of digital devices.-Donetsk:IAMM NASU.- 2005.-436p.(in Russian).
5. M.J. O'Dare, T.Arslan. Hierarchical test pattern generation using a genetic algorithm with a dynamic global reference table//First IEE/IEEE international Conference on – Genetic Algorithms in Engineering Systems: Innovations and Applications.-No.414,12-14 September 1995.-pp.517-523.
6. B.Becker, M.Keim. Hybrid fault Simulation for Synchronous Sequential Circuits//Journal of Electronic Testing: Theory and Applications.-vol.15.-1999.-pp.219-238.
7. I. Pomeranz and S.M. Reddy, "The multiple observation time strategy". *IEEE Transactions on Computers*, vol. 41, 1992.- No. 5.- pp.627-637.
8. Prinetto P., Rebaudengo M., Sonza R.M. An Automatic Test Pattern Generator for Large Sequential Circuits based on Genetic Algorithms // Proc. Int. Test Conf. – 1994. – P.240-249.
9. D.G. Saab, Y.G. Saab, J. Abraham. CRIS: A Test Cultivation Program for Sequential VLSI Circuits // In Proc. Int. Conf. on Computer Aided Design. – 1992. – P.216-219.
10. Rudnick E.M., Holm J.G., Saab D.G., Patel J.H. Application of Simple Genetic Algorithm to Sequential Circuit Test Generation // Proc. European Design & Test Conf. – 1994. – P.40-45.
11. Muzuder P.,Rudnic E.M., Genetic algorithms for VLSI design, layout&test automation.-Prentice Hall PTR, 1999.-336p.
12. Hsiao M.S., Rudnick E.M., Patel J.H. Automatic test generation using genetically-engineered distinguishing sequences // In Proc. IEEE Test Symp. – 1996. – P.216-223.
13. Prinetto P., Rebaudengo M., Sonza R.M. An Automatic Test Pattern Generator for Large Sequential Circuits based on Genetic Algorithms // Proc. Int. Test Conf. – 1994. – P.240-249.
14. Mahfoud S.W. A comparison of parallel and sequential niching methods.//Proceedings of VI International conference on genetic algorithms.-1995.-p.136-143.
15. Corno F., Prinetto P., Rebauden M., Sonza Reorda M., Veiluva E. A PVM tool for automatic test generation on parallel and distributed

systems // Proc. Int. Conf. on high-performance computing and networking. – Milan, 1995. – P.39-44.
16. Krishnaswamy D., Hsiao M.S., Saxena V., Rudnik E.M., Patel J.H. Parallel genetic algorithms for simulation-based sequential circuit test-generation // IEEE VLSI Design Conf. – 1997. – P.475-481.
17. Abramovici M. Digital System Testing and Testable Design. – New York: Computer Science Press, 1990. – 652p.
18. Hsiao M.S., Rudnic E.M., Patel J.H. Automatic test generation using genetically-engineered distinguishing sequences // Proc. VLSI Test. Symp. – 1996. – P.216-223.
19. Corno F., Cumani G., Sonza Reorda M., Squillero G. Fully Automatic Test Program Generation for Microprocessor Cores // DATE2003: Design, Automation and Test in Europe, Munich, Germany, 2003. – P.1006-1011.