Evolutionary approach to the functional test generation for digital circuits

Y.A.Skobtsov[1], D.E.Ivanov[2], V.Y.Skobtsov[2] ,R.Ubar[3],

[1]Department of Automated Control Systems, DNTU, Artema Str. 58, 83000 Donetsk, Ukraine,
skobtsov@kita.dongu.donetsk.ua
[2]Institute of Applied Mathematics and Mechanics of NAS of Ukraine,
R.Luxemburg Str. 74, 83114 Donetsk, Ukraine
{ivanov, skobtsov}@iamm.ac.donetsk.ua
Computer Engineering Department, TTU, Raja 15, 12618 Tallinn, Estonia
raiub@pld.ttu.ee

Abstract

In the paper an evolutionary approach for functional testing of digital circuits is considered. A genetic algorithm for testing digital multiplier is proposed. The main target of the proposed method is to generate as short as possible functional test wih as high as possible fault coverage with the goal to use the generated patterns as the input data for embedded functional BIST. Experimental data of the program realization is also represented.

1. Introduction

During the last thirty years of XX century, the ideas of evolution theory, self-organization and genetics developed for biological systems, were extended to the technical systems, hardware and software. The new direction in the theory and practice of artificial intelligence, called evolutionary computations, rapidly progresses now. The term *evolutionary computations* is applied to determination of the search, optimization or learning algorithms based on some formalized principles of natural evolutionary selection. Evolutionary computations use various models of the evolutionary process, which are differed by diverse representations of solutions and genetic operators. This approach is successfully applied for digital circuits testing [1,2]. In the current paper the evolutionary approach to functional testing of digital circuits is considered.

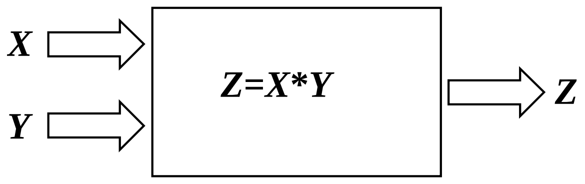


Figure 1: Multiplier

Let us consider functional test generation for the combinational circuit of multiplier (Fig.1). Suppose that the operands *X*, *Y* and the result *Z* are integer numbers for simplicity (additional information about the multiplier circuit may be introduced). The task consists in functional test generation of the minimal length desirable. A genetic algorithm is applied for solving this problem.

2. Genetic algorithm for functional test generation

Genetic algorithms (GA) are one of the evolutionary computations paradigms. They are search algorithms based on the principles, which are similar to the natural selection principles [3]. GA use a random directional search to construct (sub)optimal solution of the given problem. A subset of points (which are potential problem solutions) is chosen from the search space. This subset is called a population in terms of the natural selection and genetics. Each potential solution of the problem – individual is presented by chromosome – some gene structure. An arbitrary gene of chromosome takes a value from some alphabet that encodes the points in the search space. In the simplest case, an individual can be represented by binary encoded string (for example 0011101). It makes the GA attractive for solving the problem of logic circuits test generation, where the solution is presented as a set of binary patterns or sequences of binary patterns. A fitness function is determined on the solution set. It allows to estimate the closeness of each individual to the optimal solution – the ability of survival. The genetic search of solution consists in the simulation of such artificial population evolution. Creation of new individuals during the population evolution is based on the reproduction process simulation. In this case the individuals-solutions involved in reproduction process are called the parents, contrary the obtained individuals-solutions are called the offsprings. In each generation a set of individuals-solutions is constructed using the parts of individuals-parents and adding new parts with “good properties”. Thus the GA effectively uses the information accumulated during evolution process.

For solving any problem with genetic algorithm, first of all, we have to define: 1) the form of individual representation – encoding; 2) genetic operators – crossover and mutation; 3) fitness function (Fig.2). Further we shall consider all these factors with reference to the given problem.

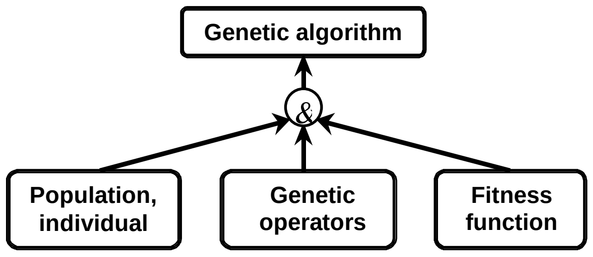


Figure 2

In the following we will consider the definition of all GA components with the reference to the formulated problem.

1) A single test pattern consists of two integer numbers *X* and *Y* – the operands of multiplier. Therefore a decision is represented as two-component vector (*X*,*Y*). On the other hand, we will use binary strings – codes of the numerical vectors, as decision representation.

2) Two types of genetic operators (crossover and mutation) are used – arithmetical and binary.

In accordance with arithmetical *crossover* for two parents $\boldsymbol{A} = (X_a, Y_a)$ and $\boldsymbol{B} = (X_b, Y_b)$, a new individual-offspring $\widetilde{\boldsymbol{A}}$ is defined as follows

$$\widetilde{X} = (1-\alpha) * X_a + \alpha * X_b,$$

$$\widetilde{Y} = (1-\alpha) * Y_a + \alpha * X_b,$$

where $\alpha \in [0;1]$.

Since we consider integer operands of multiplier and the described numerical crossover can generate real numbers, then we have to round off the generated components of the vector $\widetilde{\boldsymbol{A}} = (\widetilde{X}, \widetilde{Y})$.

Binary crossover is executed according to the classical scheme represented below in Fig. 3

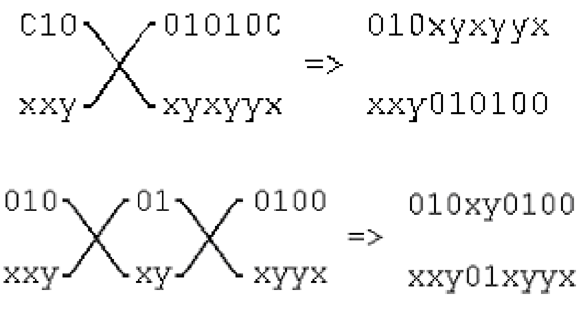


Figure 3: One-point and two-point binary crossover operators

At the same time each type of crossover is applied with its own probability P_b^c (binary) and P_n^c (numerical): $P_b^c + P_n^c = 1$. Note, that both types of crossover are used as it is shown in the algorithm pseudocode (Fig.5).

For *mutation operators* we also use two types – arithmetical and *binary* mutation. Arithmetical mutation for functional testing is implemented as follows. New individual $\boldsymbol{A}' = (X_a', Y_a')$ -«mutant» is obtained from old individual $\boldsymbol{A} = (X_a, Y_a)$ according to expressions

$$X_a' = X_a \pm \Delta * X_a,$$

$$Y_a' = Y_a \pm \Delta * Y_a$$

where Δ - is small number. Obtained individual ***A'*** must be rounded off also.

Binary mutation is implemented in accordance with the traditional method (Fig.4)

$$0\,1\,0\,0\,\boxed{1}\,0\,1\,1 \rightarrow 0\,1\,0\,0\,\boxed{0}\,0\,1\,1$$

Figure 4: Binary mutation

At the same time each type of mutation is applied with its own probability P_b^m (binary) and P_n^m (numerical): $P_b^c + P_n^c << 1$.

3) At the preliminary stage the test pattern quality is evaluated in the following way. The number of inverted bits in the multiplication result is estimated for each bit inversion in the current test pattern. The experiments have shown that a test pattern, where any bit inversion will lead to at least one bit inversion in the multiplication result, can be always found.

Our goal is now to generate test patterns such that the bit inversions of input operands produce maximum bit inversions in the multiplication result (it would be desirable to have all bit inversions). Thus we can define the matrix *P* of dimension $[2N \times M]$, where $p_{ij} = 1$, if *i*-th input bit inversion produces *j*-th output bit inversion. The matrix *P* is defined in the following way. First, all the matrix cells are zeros. Next, in the selected input pattern every bit is inverted. The matrix cells p_{ij} are defined in accordance to the produced bit inversions in the output pattern. The fitness function is defined as

$$h(\boldsymbol{A}) = \frac{\sum_{i=1}^{2N} \sum_{j=1}^{M} p_{ij}}{2NM}.$$

Further the genetic algorithm of functional test generation is considered, its pseudocode is represented in Fig. 5. The test generation of single patterns is executed in `AddBestInputToTest()function.` After the next iteration of new population generation an input test sequence is changed. If appending the best individual of the population leads to increasing test sequence fitness-function, then it is included to test.

The main target of the proposed method is to generate as short as possible functional test wih as high as possible fault coverage with the goal to use the generated patterns as the input data for embedded functional BIST [5]. The final fault coverage of the BIST can be increased to 100% by improving the controllability and observability of the circuit.

```
GA(PopulationSize,MaxIteration)
{
  // Prepare start population
  GenerateStartPopulation(PopulationSize);
  PopulationNumber=0;
  // Main GA loop starts here
  while(Stop Criterion is not reached)
  {
    NewPosition=0;
    for( int i=0 ; i<PopulationSize ; ++i)
    {
      // Do select scheme here
      DoSelect(ParentA,ParentB);
      // Do crossingover scheme here

DoCrossingover(ParentA,ParentB,Offspring);
      // Do mutation here
      DoMutation(Offspring);
      // Add new individual
         to intermediate population
      AddToIntermediatePopulation(Offspring,
      NewPosition);
      ++NewPosition;
    }
    ConstructNewPopulation(PopulationSize);
    AddBestInputToTest();
    // increase the population number
    ++PopulationNumber;
  }
  // report the results both screen & file
  DoReport();
} // end GA
DoSelection(ParentA,ParentB)
{
  EvaluateFitness(Population,PopulationSize,
  Fitness);
  DoRouletWheelSelection(ParentA,ParentB,
  Population,PopulationSize,Fitness);
}
DoMutation(Offspring)
{
  // select with small probability – do
     mutation or not
  if(NeedMutation())
  {
    // select the mutation sheme
    SelectMutationSheme();
    if( FuncionalMutation )
    {
      DoFunctionalMutation(Offspring);
    }
    else
    {
      DoBinaryMutation(Offspring);
    }
  }
}
DoCrossingover(ParentA, ParentB, Offspring)
{
  // select the crossingover sheme
  SelectCrossingoverSheme();
  if( FunctionalCrossingover )
  {
    DoFunctionalCrossingover(ParentA,ParentB,
    Offspring);
  }
  else
  {
    DoBinaryCrossingover(ParentA,ParentB,
    Offspring);
  }
}
ConstructNewPopulation(PopulationSize)
{
  // construct temporary population
  CombinePopulations(Population,
  IntermediatePopulation);
  // sort temporary population in ascending
     value of fitness
  SortCombinedPopulationByFitness();
  // select as next population top of
     temporary population
  CopyTopOfPopylation(PopulationSize);
}

DoFunctionalMutation(Offspring)
{
  SelectSmallDelta();
  Offspring=Offspring±Delta*Offspring;
}
DoFunctionalCrossingover(ParentA,ParentB,
Offspring)
{
  SelectSmallAlpha();
  Offspring =(1-Alpha)*ParentA+Alpha*ParentB;
}
```

Figure 5: GA pseudocode

3. Experimental results

The suggested algorithm was implemented in C++ language in C++ Builder environment. Experimentally there were defined the values of crossover and mutation probabilities $P_c = 0.8$ and $P_m = 0.01$, coefficients $\alpha = 0.5$ and $\Delta = 0.5$ for functional crossingover and mutation accordingly, the individuals number in population – 100. Under these conditions the dependence of test fitness-function (fault coverage) from generations number was investigated. The experimental data in Fig. 6 show that fitness-function value is stabilized enough quickly. Hence the boundary value of GA generations number is chosen equal to 40.

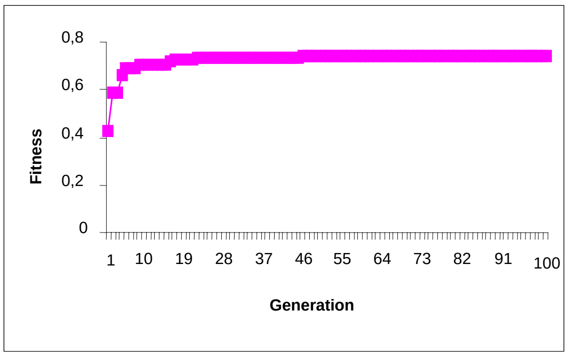
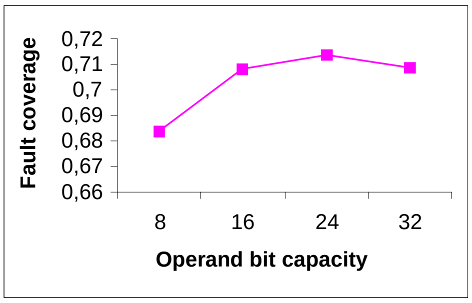
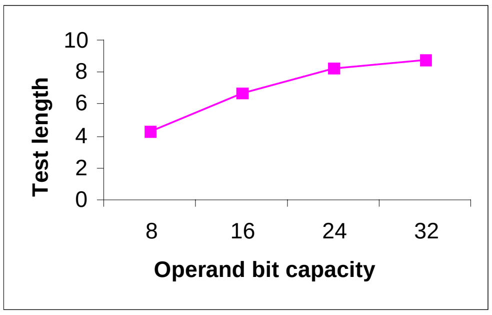


Fig.6. Growth of fitness-function value depending on generation number



a)



b)

Fig.7. Dependence of test fault coverage and test length from operand bit capacity.

Then the results of the experiments to show the dependence of test fault coverage and test length on the operand bit capacity are shown in Fig.7. Average results of 10 experiments are cited. In Table 1, an example of the functional test for the 32-bit multiplier is presented, which consists of 11 patterns..

4. Conclusion

The test generation for 32-bit multiplier, in accordance with considered approach, shows that functional tests with 70% coverage relatively to primary input bits inverting were generated enough quickly. The test length is equal to 10 patterns for 32 primary inputs. Thus an evolutionary approach to functional test generation, at the example of multiplier, was investigated and proposed.

The main target of the proposed method is to generate as short as possible functional test to reduce the amount of input data for embedded functional BIST. The final fault coverage of the BIST can be increased to 100% by improving, correspondingly, the controllability and observability of the circuit.

Acknowledgements. This work has been supported by EU V Framework projects REASON and EVIKINGS, as well as by the Estonian Science Foundation grants 5649 and 5910

References

[1] P. Prinetto, M. Rebaudengo, M. Sonza Reorda, "An Automatic Test Pattern Generator for Large Sequential Circuits based on Genetic Algorithms" In Proc. Int. Test Conf., 1994, pp. 240-249.

[2] E.M. Rudnick, J.H. Patel, G.S. Greenstein, T.M.Niermann, "Sequential Circuit Test Generation in a Genetic Algorithm Framework". In Proc. Design Automation Conf., 1994, pp.40-45.

[3] D.E. Goldberg, Genetic Algorithms in Search, Optimization & Machine Learning. Addison-Wesley Publishing Company, Inc., 1989.

[4] M.Abramovici, M.A.Breuer, A.D.Friedman, Digital systems testing and testable design. IEEE Press Inc., New York.

[5] R.Ubar, N.Mazurova, J.Smahtina, E.Orasson, J.Raik. HyFBIST: Hybrid functional built-in self-test in microprogrammed data-paths of digital systems

Table.1 Example of the functional test for 32-bit multiplier

Decimal representation		Binary represetation	
A	B	A	B
1373907781	127562156	01010001111001000010101101000101	01001100000010000111000010111001
-254785975	376343075	11110000110100000100011001001001	00010110011011101000101000100011
1630710233	579247396	01100001001100101010100111011001	00100010100001101001110100100100
1373907165	1277006115	01010001111001000010100011011101	01001100000111011001000100100011
13649481	1382976035	00000000110100000100011001001001	01010010011011101000101000100011
67977233	70853037	00000100000011010100000000010001	00000100001110010010000110101101
122962474	458015755	00000111010101000100001000101010	00011011010011001100010000001011
1376166070	419570069	01010010000001101010000010110110	00011001000000010001000011001010 1
92684874	1309303734	00000101100001100100001001001010	01001110000010100110001110110110
26034540	646202154	00000001100011010100000101101100	00100110100001000100001100101010