# Efficient Evolutionary Approach for the Test Compaction Problem

Doina LOGOFĂTU
*University of Applied Sciences,*
*Faculty of Computer Science and Mathematics,*
*Lothstr. 64, 80335 Munich, Germany*
*doinabooks@yahoo.com*

***Abstract*—The test compaction is one of most important requirement regarding the large scale integration (LSI) testing. The overall cost of a VLSI circuit's testing depends on the length of its test sequence; therefore the reduction of this sequence, keeping the coverage of error prone points, will lead to a reduction of used resources in the testing process. This problem is NP-complete. Consequently an optimal algorithm doesn't have applicability in practice. In this paper we describe an evolutionary algorithm (GATC) and we introduce the term of compaction factor (cf), i.e. the "expected" percentage of compacted test sequence. GATC provides in praxis better results than a greedy approach (GR) for many configurations. This improvement comes from the freedom to merge randomly pairs of compatible tests for different candidates to solution and keeps the ones with more "Don't care" positions, thus there is an increased probability to find for them compatible tests in the next stage. Also the C++ implementation was optimized, using compact data structures and the Standard Template Library.**

***Index Terms*—Evolutionary Algorithms, Digital Circuit Design, Test Compaction Problem, Set Coverage Problem, Test Generation, Greedy Algorithm, Optimization, Don't Care Value**

## I. INTRODUCTION

The overall testing process is very important for a circuit and one issue is minimizing the cost of testing by finding effective test sequences with a minimal size, which cover all faults (see e.g. [10, 13]). The size of the test sequence directly affects the overall cost of the circuit testing. The test compaction is the activity to reduce the length of this sequence, by keeping its effectiveness. This will lead to a testing time which consumes no more than necessary time resources. In praxis, the test compaction for a VLSI circuit is done in two ways: dynamically and statically. The dynamic test compaction is performed simultaneous with the test generation and the static test compaction is used after the test generation, as an optimization step.

In this paper we propose an evolutionary approach for the Test Compaction Problem, which could be used, also combined with other techniques, in the static compaction phase. Recently, EAs have been successfully applied to several problems in VLSI CAD (see e.g. [8, 9, 20, 25]). For specifically test compaction issues there are also proposed more methods (see e.g. [10, 13]). We transform the general problem in a particular one and provide the formal framework in Section 2: the problem domain, description of optimal and greedy methods. Based on experimental results, we concluded that the Greedy method leads to very good quality results. In section 4 we present an evolutionary algorithm, which provides in the most cases better results as Greedy. For test generation we use a test generator module, capable to produce artificial benchmarks with an expected compaction rate (the expected percentage of the optimal compacted sequence, which covers the input sequence). As showed in our experiments, this factor influences the results and could be used to develop different kind of methods (from both design and implementation point of view). The paper contains 3 tables, 5 figures and an end section with a conclusion and various ideas to continue the research.

## II. PROBLEM DOMAIN AND CLASSICAL APPROACHES

Let us consider a set of test sequences $T = \{S_1, S_2, \ldots, S_n\}$ detecting (covering) the set of faults $\{f_1, f_2, \ldots, f_m\}$ of a sequential circuit. Every test sequence $S_i = \{v_1, v_2, \ldots, v_{ii}\}$, $i = 1, \ldots, n$ is an ordered set of $L_i$ test vectors $v_1, v_2, \ldots, v_{Li}$, where $L_i$ is the length of $S_i$. A fault $f_i$ within a sequence $S_j$ has the detection cost $d_{ij}$ equal to the number of vectors from the beginning of the sequence until $f_i$ becomes detected in $S_j$. Test compaction problem is to find a collection of subsequences, i.e. subsets of vector sequences, so that all faults in $F$ are covered and the test length of the collection is a minimum. This problem can be reformulated as a set covering problem, which is NP-complete. Further we will reformulate and specialize it, using a concrete number of faults.

We represent the tests as strings, which contain only characters from a set $S$ with cardinal number 5 and one of this character is 'Don't Care' ($X$). This means that it is compatible with any other character from $S$. We consider that two characters are compatible if they are the same or one of them is Don't Care. For example, the tests $t_1$=10ZX0XU and $t_2$=X0Z10UU are compatible because all same position pairs from them are compatible (1-X, 0-0, Z-Z, X-1, 0-0, X-U and U-U). Two compatible tests can be reduced to one test, by using a merge operation on every position in both of them: one character merged to it gives itself, one character merged to $X$ gives this character. For our example: Merge ($t_1$, $t_2$) = 10Z10UU. The problem is to reduce a given sequence of tests to a sequence of minimal length, by successively applying the Merge operation.

An optimal method has to explore the whole space of solutions and pick one with minimal length. This exhaustive search has an exponential complexity and no applicability in practice; it can process only very small instances of the

problem. Other approach is a Greedy one, by applying repeatedly the Merge operation on first found compatible pair of tests and modifying the current sequence. This algorithm is polynomial and can cope with large data sets in an acceptable execution time. From our experiments results that, for relative small data (various parameters: 25 to 30 tests, every with a length between 20 and 40), the greedy algorithm leads in the most cases to the same result as the optimal one. From 625 cases, only in 10 cases we obtained with Greedy a sensible lower quality of results and this is only 1.6%. For this experiment we generated artificial input sequences, which had a compaction rate of around 50%. This allowed us to observe the behavior of two methods for sequences which are 'compressible' and conclude that the Greedy method is a powerful instrument regarding both quality of the results and execution time.

## III. PREVIOUS APPROACHES

This problem is very similar to the well-known *Set Coverage Problem*. This is contained in the list of 21 classical NP-complete problems, for which Richard Karp demonstrated that they are included in the class of NP-complete problems. It has a very wide area of applicability, in domains like: wireless sensor networks (*k-Sensor Coverage Problem*, see e.g. [2]), urban systems (see e. g. [25]), VLSI Test Coverage (see e.g. [14, 17]), discrete geometry (e.g. image processing, covering for hyper cubes).

Some useful ideas regarding the test compaction process are presented in [14], where the authors introduce a method based on fault simulation, and they present some heuristics for reducing the simulation effort. The test sequences are fully specified, generated by a sequential ATPG and the proposed methods produce a test sequence which contain lots of Don't Cares (*X*s) without losing stuck-at fault coverage of the original test sequence. Many greedy approaches with an accurate analysis for the (*k*-) *Set Coverage Problem* were developed (see e.g. [15]) and were proposed some formal different evolutionary approaches for variations of the problem (see e.g. [1, 17]).

## IV. EVOLUTIONARY ALGORITHM

We use a genetic algorithm, which transforms the population along a number of generations. Every individual will be a sequence of tests, which is overall compatible with the input sequence and thus a possible solution. In a specific generation, the next population is constructed on the current one, by preserving a part of individuals and filling the rest with copies of some of the best individuals. The fitness-function in this stage is defined as the overall number of Don't Cares in sequence. A higher value of this function for an individual increases the probability to find afterwards compatible pairs. At the end, every individual in population will be compacted by using the greedy algorithm.

### *A. Overview of Genetic Algorithms*

A Genetic Algorithm (GA) is an optimization method with simple operations based on the natural selection model [26]. The genetic algorithms have been applied to hard optimization problems including VLSI layout optimization, boolean satisfiability and the *Set Cover Problem* (see e.g. [9, 16, 17, 25]). There are four main distinctions between GA-based approaches and traditional problem-solving methods:

a) GAs operate with a genetic representation of potential solutions, not the solutions themselves.
b) GAs search for optima of a population of potential solutions and not a single solution (the genetic operators alter the composition of children).
c) GAs use evaluation functions (*fitness*), no other auxiliary knowledge such as derivative information used in the conventional methods.
d) GAs use probabilistic transition rules (not deterministic rules) and various parameters (population size, probabilities of applying the genetic operators, etc.)

For a specific problem, it is very important to use related genetic operators, which preserve the good traits from the parents, but are also able to bring improvements in the resulting children. The initialization step and the parameter settings are also very significant. Often, the GAs are used mixed with another programming techniques, for example greedy for generating good start individuals. In this case they are called hybrid GAs.

```
ALGORITM_GA
  Initialise_Random_Population()
  While (not terminal case) Execute
      ApplyGeneticOperators();
      CalculteFitnessForAllIndividuals();
      UpdatePopulation();
  End_While
END_ALGORITHM_GA
```

**Figure 1.** Pseudo code for a Genetic Algorithm.

The terminal case could be a specific condition which should be satisfied from the population or from the best individual (if it represents an acceptable solution for the problem).

### *B. A Genetic Algorithm (GATC) for the Test Compaction Problem*

This algorithm is based on the classical sketch of a genetic algorithm (figure 1), where the individuals in the initial population are copies of the start sequence. Mutations are applied on the current population and the best individuals are kept for the next iteration. After a number of iterations, we apply on every individual the Greedy algorithm (described in section 3) in order to obtain a compacted coverage set of tests. The particularity of this approach is the initialization with copies of the start sequence and the usage of the Greedy approach during the final phase.

*Representation.* A potential solution is a coverage set of the input sequence, i.e. a set which contains a compatible test for any test contained in the input sequence. A population is a set of such elements. On its individuals are applied a succession of mutation operators. A mutation operator is a substitution of two compatible tests with their merged one.

*Initialization.* Often it is helpful to combine EAs with

problem-specific heuristics (see e.g. [5, 8, 9, 20, 24]) and the initial population contains a number of individuals which are enhanced by using other techniques, like e. g. Greedy. In our case, we will not use any specific techniques for initialization. The initial individuals are copies of the input sequence. They will differentiate themselves by applying the mutation operators.

*Objective Function and Selection.* We will use as fitness function the total number of Don't Cares ('X's):

$$N_X(t_1, t_2, ..., t_n) = \sum_{i=1}^{n} \#X(test_i) \quad (1)$$

where *#X*() is a function which gives the number of 'X' characters in the parameter (which is a test string). In the selection phase we consider that an individual with a higher fitness is better than other with a lower one. We keep for the next iteration half from the best individuals and the rest are copies of some of them (for two identical tests the mutation operator will lead fast always to different individuals).

*Algorithm.* A refined version of the classical genetic algorithm:

```
ALGORITHM_GATC
    initialize(populationSize)
    initialize(mutationRate)
    numMutations ← populationSize*mutationRate
    initialize(individuals)
    For (i ← 1; i ≤ numGenerations; step 1)
      apply_Mutation_Operators(numMutatios);
      calculate_Fitness(allNewIndividuals);
      remove_Worst_Inviduals (populationSize/2);

complete_With_Copy_Individuals(populationSize/2)
      return best_element(individuals);
  END_ALGORITHM_GATC
```

**Figure 2.** Pseudo code for the TCP's hybrid EA.

*Parameter Settings.* The chosen settings are based on experimental tests. Since the genetic algorithm is applied to different data sizes, from very small to large ones, it becomes necessary to adapt these settings to the size of the problem. In our case, the necessary time to create and process a new generation for large data sets is very high. Therefore the number of generations is also related to the input data size.

## V. EXPERIMENTAL RESULTS

Like in the experiments from above, a large amount of test cases with different parameters were generated: same number of tests and different lengths with same expected compaction rate or same dimensions with different compaction rates. In many of them, GATC provided higher quality results as GR. We concentrate only of the quality of results and not the execution time: the experiments shows better ones as the GR results, by a considerable increase of execution time with the dimension of the input data. A fragment of these results are further presented and commented in tables 1, 2, 3. In the next tables, *#tests* denotes the number of tests in the initial sequence, *#length* the length of every test, output columns *%* denote the compaction rate (how small is the result sequence compared to initial one) and output columns *sec* the execution time in seconds for the solved instance.

TABLE I
COMPARATIVE RESULTS FOR ALGORTIHMS GATC AND GR: SIZE BETWEEN 100 AND 300, LENGTH OF THE TESTS BETWEEN 60 AND 100, EXPECTED COMPACTION RATE 20% (FIRST 9 CASES), RESPECTIVELY 50% (CASES 10-18). NOTE: THE SAME RESULTS PROVIDED BY BOTH ALGORITHMS ARE MARKED WITH BOLD

| **Input Data** | | | **GR** | | | |
|---|---|---|---|---|---|---|
| #case | #tests | #length | % | sec | % | sec |
| 1 | 100 | 60 | 16.00 | 0 | 15.00 | 1 |
| 2 | 100 | 80 | 24.00 | 0 | 20.00 | 1 |
| 3 | 100 | 100 | **9.00** | **0** | **9.00** | 2 |
| 4 | 200 | 60 | 27.50 | 0 | 23.00 | 12 |
| 5 | 200 | 80 | 20.50 | 0 | 16.50 | 4 |
| 6 | 200 | 100 | 24.00 | 0 | 23.00 | 12 |
| 7 | 300 | 60 | 20.67 | 0 | 20.00 | 15 |
| 8 | 300 | 80 | 25.00 | 1 | 24.00 | 17 |
| 9 | 300 | 100 | 25.33 | 1 | 23.33 | 20 |
| 10 | 100 | 60 | 58.00 | 0 | 54.00 | 3 |
| 11 | 100 | 80 | 63.00 | 0 | 61.00 | 3 |
| 12 | 100 | 100 | 53.00 | 0 | 52.00 | 3 |
| 13 | 200 | 60 | 49.50 | 1 | 47.00 | 22 |
| 14 | 200 | 80 | **53.50** | **1** | **53.50** | 21 |
| 15 | 200 | 100 | 60.50 | 2 | 59.50 | 21 |
| 16 | 300 | 60 | 51.67 | 3 | 50.67 | 65 |
| 17 | 300 | 80 | 58.67 | 4 | 56.67 | 71 |
| 18 | 300 | 100 | 57.00 | 5 | 54.33 | 81 |

For a large amount of generated test sequences with a size between 100 and 500, every test with a length between 50 and 1000, we obtained better results using GATC in over 91% of cases. For identical dimensions, the results are better for cases when the expected compaction rate is smaller (e. g. better results for an expected compaction rate around 20% as for one around 60%, identical dimensions). For a fixed number of tests, the achievements provided with GATC are better for a smaller length of them (e. g. for number of tests 100, GATC provided much better results for a length 50 as for the length 1000).

Table 2 contains results provided by GR and GATC for larger input data: our experiments showed that the GATC algorithm leads to better results in over 83% from cases with this dimension. The expected compaction rate was kept to 30% during our experiments. A visual expression of Table 3 can be seen in the Figure 5 on the last page of this paper.

TABLE II
COMPARATIVE RESULTS FOR ALGORTIHMS GATC AND GR AND LARGE DATASETS: SIZE BETWEEN 500 AND 1500, LENGTH OF THE TESTS BETWEEN 1000 AND 7000, EXPECTED COMPACTION RATE 30%

| **Input Data** | | | **GR** | | | |
|---|---|---|---|---|---|---|
| #case | #tests | #length | % | sec | % | sec |
| 1 | 500 | 1000 | 37.40 | 8 | 36.00 | 164 |
| 2 | 500 | 4000 | **30.40** | 15 | **30.40** | 303 |
| 3 | 500 | 7000 | 33.40 | 24 | 33.20 | 529 |
| 4 | 1000 | 1000 | 32.90 | 56 | 32.40 | 1114 |
| 5 | 1000 | 4000 | 30.10 | 136 | 29.90 | 2485 |
| 6 | 1000 | 7000 | 30.00 | 243 | 29.70 | 5625 |
| 7 | 1500 | 1000 | 36.60 | 195 | 35.67 | 3897 |
| 8 | 1500 | 4000 | 30.93 | 483 | 30.73 | 9342 |
| 9 | 1500 | 7000 | 30.33 | 679 | 30.20 | 13693 |

Further, we kept the same dimension of input data and changed only the expected compaction to one higher as 65%. From 600 instances generation and randomly executions of this diagnosis, the results were in over 93% the same. The execution time for this kind of inputs data was sensibly the same with the corresponding test from Table 2; this means that the expected compaction rate doesn't influence the execution time of both algorithms GR and GATC for the same dimension of inputs.

Another experiment was to keep the number of tests in sequence and to decrease significantly the length of tests. In this case the GATC provided always better results (see table 3 and figure 5 on the last page).

TABLE III
COMPARATIVE RESULTS FOR ALGORTIHMS GATC AND GR AND LARGE DATASETS: SIZE BETWEEN 500 AND 1500, LENGTH OF THE TESTS BETWEEN 10 AND 110, EXPECTED COMPACTION RATE 30%

| **Input Data** | | | **GR** | | | |
|---|---|---|---|---|---|---|
| #case | #tests | #length | % | sec | % | sec |
| 1 | 500 | 10 | 37.00 | 0 | 35.20 | 3 |
| 2 | 500 | 60 | 40.00 | 2 | 38.20 | 38 |
| 3 | 500 | 110 | 39.80 | 1 | 38.00 | 56 |
| 4 | 1000 | 10 | 37.70 | 1 | 36.60 | 16 |
| 5 | 1000 | 60 | 42.70 | 9 | 42.00 | 199 |
| 6 | 1000 | 110 | 41.10 | 12 | 39.90 | 245 |
| 7 | 1500 | 10 | 37.60 | 3 | 37.33 | 49 |
| 8 | 1500 | 60 | 43.60 | 33 | 41.67 | 671 |
| 9 | 1500 | 110 | 40.47 | 61 | 39.67 | 1205 |

An explanation of this effect is the fact that in this case the space of solutions has a considerable undersize related to the space of solution for the similar cases in table 2 (with the same initial size of the sequence but longer tests). It would be expected that the algorithm provides, naturally, even better results for a bigger population and/or more generations also for the instances from table 2.

## VI. CONCLUSIONS AND FUTURE WORK

All algorithms are implemented in C++ using the *Standard Template Library* (STL). Because a test sequence can contain only 5 characters, we represented each of them with a code of fixed length, on 3 bits: '0' – 000, '1' – 001, 'U'- 010, 'Z'-011 and 'X' – 111 and we can use the *std::bitset*, which contains all operations needed for bit strings.

After a formal description of the problem, we described an optimal solution, for that the complexity is exponential and it can be used for small input data. Additionally, we described also a Greedy approach (GR), which is in practice efficient for large data size: it provides excellent quality results in acceptable execution time.

It follows an accurate description of a proposed genetic algorithm (GATC): representation and initialization of individuals/population; fitness-function and selection; pseudo code for the GATC and experimental results for different categories of input data. Table 1 presents comparative results GR vs. GATC for relative small data sets (size 100 to 300, test length 60 to 100): for most of cases GATC provides higher quality results, which can be also seen visually in figure 3. Table 2 presents comparative results of GR vs. GATC for large data sets: size 500 to 3000, test length 1000 to 7000. Also for them, in the most cases GATC leads to better compaction rates, that means smaller coverage sets.

The experiments showed that the behavior of algorithms changes by varying different parameters: size of the input data, length of a test, expected compaction rate. The results quality provided by GATC can be increased by improving the parameter settings or adapt them specifically to the input data traits. Also improvements can be done by the implementation details; for example, experiments showed that a *STL std::string* representation will lead to faster execution times for some very specific inputs (e.g. size 100 to 500, length of tests 5000 to 10000, expected compaction rate 20%). Also an analysis of the expected compaction rate can be useful, our experiments showed different quality of results for different expected compaction rates. This kind of diagnosis and improved hybrid GAs will lead to faster and higher quality solutions, capable to cope with larger data sets. Another direction could be the classification of the tests with more Don't Cares on the pretty same positions (or other classification criteria) and the split of the input sequence in more subsequences (classification classes); to solve them and then combine their results (a kind of *divide-et-impera* technique). Developing more genetic operators for individuals and applying them in combination with the proposed mutation operator could lead also to improvements of the results.

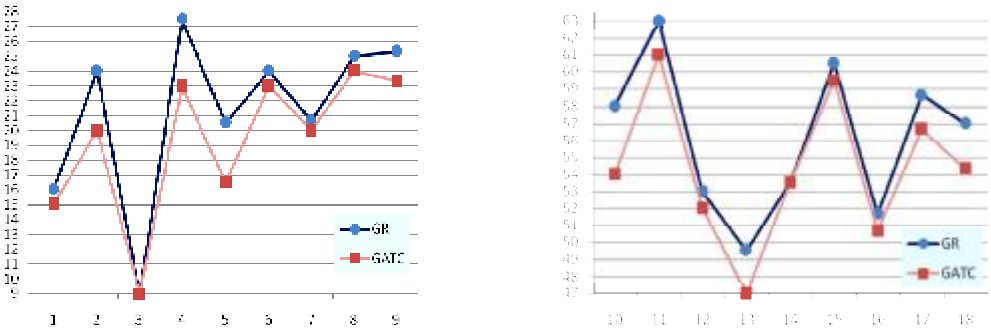


**Figure 3.** GR and GATC compaction percentage for the test cases 1-9, respectively 10-18, Table 1
*#Test cases / Percentage* (GATC percentage is fast always under GR percentage).

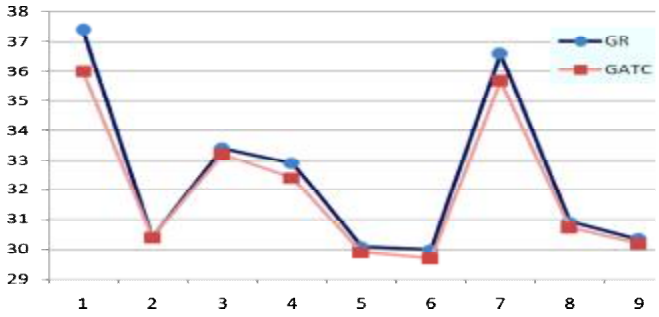


**Figure 4.** GR and GATC compaction percentage for Table 2.

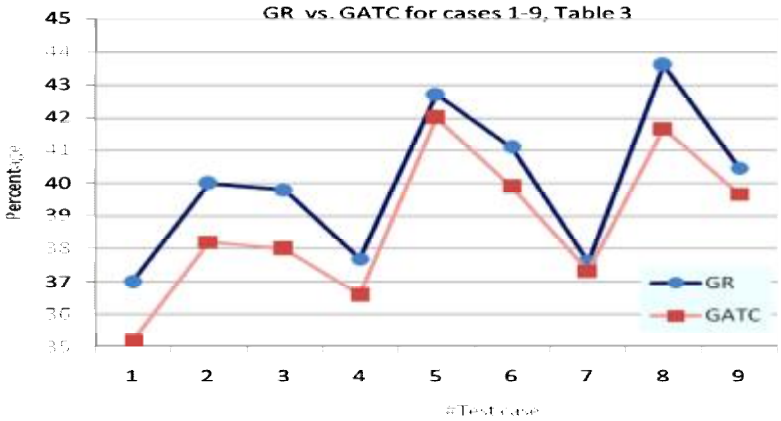


**Figure 5.** GR and GATC compaction percentage for Table 3.

## REFERENCES


[1] Alon, N., Moshkovitz, D., Safra, M.: "Algorithmic construction of sets for k-restrictions," *ACM Transactions on Algorithms (TALG)*, v. 2 n.2, pp. 153-177, 2006.

[2] Cardei, M., Wu, J.: "Energy-efficient coverage problems in wireless ad-hoc sensor networks," *Computer Communications*, v. 29 n. 4, pp. 415-420, 2006.

[3] Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C.: *Introduction to Algorithms*, 2nd Edition, MIT Press, 2001.

[4] Davis, L.: "Applying adaptive algorithms to epistatic domains," In *Proceedings of IJCAI*, pages 162-164, 1985.

[5] Davis, L.: *Handbook of Genetic Algorithms*, New York, 1991.

[6] De Micheli, G.: Synthesis and Optimization of Digital Circuits. McGraw-Hill, Inc., 1994.

[7] Drechsler, N., Drechsler, R.: Exploiting don't cares during data sequencing using genetic algorithms. In *ASP Design Automation Conf.*, pp. 303-306, 1999.

[8] Drechsler, R., Drechsler, N.: Evolutionary Algorithms for Embedded System Design. Kluwer Acadmeic Publisher, 2002.

[9] Drechsler, R.: Evolutionary Algorithms for VLSI CAD. Kluwer Academic Publisher, 1998.

[10] El-Maleh, A., Osais, Y.: Test vector decomposition based static compaction algorithms for combinatorial circuits, *ACM Trans. Des. Autom. Electron. Syst.*, vol. 8, pp. 430-459, 2003.

[11] Feige, U.: A Thereshold of lnn for Approximating Set Cover, *Journal of the ACM (JACM)*, v. 45 n. 4, pp. 634-652, 1998.

[12] Garey, M. R., Johnson, D. S.: Computers and Intractability – A Guide to NP-Completeness. Freeman, San Francisco, 1979.

[13] Guo, R., Pomeranz, I., Reddy, S. M., On improving static test compaction for sequential circuits, *VLSI Design*, Fourteenth International Conference, pp. 111-116, 2001.

[14] Higami, Y., Kajihara, S., Pomeranz, I., Kobayashi, S., Takamatsu, Y.: On Finding Don't Cares in Test Sequences for Sequential Circuits, *IEICE Transactions on Information and Systems*, v. E89 n. 11, 2006.

[15] Hochbaum, D., S., Pathria, A.: Analysis of the greedy approach in problems of maximum k-coverage, *Naval Research Logistics*, v. 45 n.6, pp. 615-627, 1998.

[16] Holland, J. H.: Adaption in Natural and Artificial Systems. The University of Michigan Press, Ann Arbor, MI, 1975.

[17] Ibrahim, W., El-Chouemi, A., El-Sayed, H.: Novel Heuristic and Genetic Algorithms for the VLSI Test Coverage Problem, Computer Systems and Applications, *IEEE International Conference*, pp. 402-408, 2006.

[18] Karp, M. R.: Reductibility Among Combinatorial Problems, *Complexity of Computer Computations* (Symposium Proceedings), Plenum Press, 1972.

[19] Logofătu, D.: Algorithmen und Problemlösungen mit C++, Vieweg-Verlag, 2006.

[20] Logofătu, D.: Drechsler, R., Efficient Evolutionary Approaches for the Data Ordering Problem with Inversion, 3rd *European Workshop on Hardware Optimisation Techniques (EvoHOT)*, LNCS 3907, pp. 320-331, Budapest, 2006.

[21] Logofătu, D.: Greedy Approaches for the Data Ordering Problem with Inversion, *Proceedings of ROSYCS*, Romanian Symposium on Computer Science, pp. 65-80, Iași, 2006.

[22] Logofătu, D.: Algoritmi fundamentali in C++. Aplicații, Editura Polirom, Iași, 2007.

[23] Logofătu, D.: Algoritmi fundamentali in Java. Aplicații, Editura Polirom, Iași, 2007.

[24] Lund, C., Yannakakis, M.: On the hardness of approximating minimization problems, *Journal of the ACM (JACM)*, v. 41 n. 5, pp. 960-981, 1994.

[25] Mazumder, P., Rudnick, E.: Genetic Algorithms for VLSI Design, Layout & Test Automation. Prentice Hall, 1998.

[26] Michalewicz, Z.: Genetic Algorithms + Data Structures = Evolution Programs, 3rd ed. Springer-Verlag, Berlin Heidelberg New York (1996).

[27] Murray, A., T., Kim, K., K., Davis, J., W., Machiraju, R., Parent, R.: Coverage optimization to support security monitoring, Computers, *Environment and Urban Systems*, vol. 31, n. 2, pp 133-147, 2007.