Application of a Genetic Algorithm for Testing SEUs in SRAM-FPGA Systems

Cinzia Bernardeschi, Luca Cassano, Mario G.C.A. Cimino, Andrea Domenici

Department of Information Engineering, University of Pisa, Italy
{c.bernardeschi, luca.cassano, m.cimino, a.domenici}@ing.unipi.it

Abstract. Testing of FPGAs is gaining more and more interest because of the employment of FPGA devices in many safety-critical application fields. We propose a prototype of a tool for the generation of test patterns for application-dependent testing of SEUs in SRAM-FPGAs based on a genetic algorithm. We focus on SEUs affecting logic resources of the FPGA. SEUs in any configuration bit are addressed, making our fault model much more accurate than the classical stuck-at fault model. Results from the application of the tool to some circuits from the ISCAS and ITC benchmarks are reported.

Keywords: Genetic Algorithm; Test Pattern Generation; SEUs; SRAM-FPGAs

1 Introduction and Related Works

The industrial use of electronic devices in safety-critical systems is regulated by application-related safety standards that impose strict safety requirements to the system. In particular safety standards, such as ISO 26262-5 [8], CENELEC 50129 [7], and IAEA NS-G-1.3 [12], require in-service testing activities for safety-related systems. It is therefore vital that the tests are able to detect the largest number of faults that may occur in the system.

Radiations in the atmosphere are responsible for introducing *Single Event Upsets (SEU)* in digital devices [1]. SEUs have particularly adverse effects on FPGAs using SRAM technology, as they may permanently corrupt a bit in the configuration memory (correctable only with a reconfiguration of the device) [10].

Two main families of test methods for FPGA circuits exist: *application-dependent* and *application-independent*. Application-independent methods, such as [11, 16, 21], aim at detecting structural defects due to the manufacturing process of the chip and they are performed by the chip manufacturer. These methods are called application-independent because they target every possible fault in the device without any consideration of which parts of the chip are actually used by the given design and which parts are not.

Conversely application-dependent methods [17, 22], address faults in the resources of the FPGA chip used by the implemented application and allow in-service testing.

Since FPGAs cannot be considered as classical ASIC circuits, classical test methods for digital circuits fail when applied to FPGAs [15]. In particular, it is demonstrated that test pattern generation methods based on the stuck-at fault model for ASIC circuits obtain too optimistic results when applied to SRAM-FPGAs. The stuck-at fault model models faults at the input and output signals of the logical components, and more accurate fault models, keeping into account faults in the configuration bits of the FPGA chip, should be considered [14].

In this work, we propose a prototype of an automatic test pattern generation tool based on a *genetic algorithm* (GA) for application-dependent testing of SEUs in SRAM-FPGAs, that takes into account SEUs in configuration bits of the FPGA. In particular, we focus on SEUs affecting logic components of the FPGA leaving out SEUs affecting the routing structure, which will be object of further work. To the best of our knowledge, very few (if any) works on this kind of FPGA faults are reported in the current literature.

Genetic algorithms [9, 13] have often been used for test pattern generation for digital circuits [4, 18, 20]. The proposed GA uses the simulation-based fault injection tool for FPGAs presented in [3] to calculate the fault coverage obtained by each generated test pattern. Before this work the tool was used for fault observability and failure probability estimation with randomly generated test vectors [2].

Our tool generates a sequence of test vectors to be used as an in-service test. Such a sequence is chosen by a GA whose goal is optimizing fault coverage, keeping into account the need of limiting the test sequence length, according to the time constraints of in-service testing.

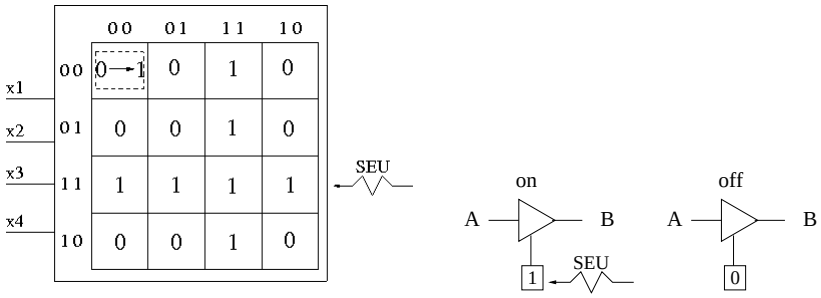
The remainder of the paper is organized as follows: in Sect. 2 the considered fault model is presented; in Sect. 3 the main characteristics of the proposed GA are discussed; Sect. 4 briefly presents the fault injection tool, leaving a more detailed description to the referred paper; Sect. 5 presents the results of the proposed algorithm for some circuits from ISCAS and ITC benchmarks; Sect. 6 concludes the paper.

2 The Fault Model

An FPGA is a prefabricated array of programmable blocks, interconnected by a programmable routing architecture and surrounded by programmable input/output blocks.

Programming an SRAM-FPGA device consists in downloading a programming code, called a *bitstream*, into its configuration memory. The bitstream determines the functionalities of logic blocks, the internal connections among logic blocks and the external connections among logic blocks and I/O pads. Interconnections are realized internally by routing switches and externally by *I/O buffers*. The commonest programmable logic blocks are *lookup tables* (LUT), small memories whose contents are defined by configuration bits.

We consider the FPGA system at the netlist-level representation produced in the synthesis phase before the place and route. At this level, the elements



(a) Lookup table failure (b) I/O buffer failure

Fig. 1. Failure modes of various resources of the FPGA chip.

visible in the model are I/O buffers, LUTs, flip-flops, and multiplexers, thus we consider SEUs in the configuration memory of LUTs and I/O buffers while we do not address faults in the routing structure of the FPGA device.

In the stuck-at fault model, a SEU in the configuration memory of a component causes the output of the faulty component to be stuck at a given value, thus the fault is always active. In the fault model considered in this work, a SEU in the configuration memory of a LUT causes an alteration of the functionality performed by the LUT and the fault is active only when the configuration of the inputs of the faulty LUT is the one associated with the faulty configuration bit. Figure 1(a) shows a SEU causing a bit flip in the configuration bit associated with input (0 0 0 0). In this case the logic function implemented by the LUT changes from $y = x_1 \cdot x_2 + x_3 \cdot x_4$ to $y_f = x_1 \cdot x_2 + x_3 \cdot x_4 + \overline{x_1} \cdot \overline{x_2} \cdot \overline{x_3} \cdot \overline{x_4}$. In the example, when the input values are (0 0 0 0) the faulty LUT behaves like y_f, meaning that the fault has been activated, otherwise it behaves like y. We observe that an n-input LUT has 2^n possible faults in the configuration bits. A SEU in the configuration bit of a buffer causes an undesired connection or disconnection between two wires, as shown in Fig. 1(b).

3 The Genetic Algorithm

Many complex problems may be solved by *search* methods, i.e., procedures that look for a solution by trying out many attempts until a satisfactory result is obtained. Such an attempt might be, e.g., a sequence of moves in a game, a set of variable assignments to solve an equation, or a set of parameter values to optimize a function. Often more than one solution exists, and some solution may be better than others according to given criteria [9].

A GA is a search method based on the analogy with the mechanisms of biological evolution. GAs require that any solution to a given problem be *encoded*, i.e., represented as a sequence of symbols, that stands for a chromosome (a sequence of genes) in the biological analogy. A GA starts from an initial set (a *population*) of tentative solutions (called *chromosomes*), *selects* the best ones

according to a problem-specific *fitness* function, and the selected chromosomes are combined and mutated to produce a new population. These operations have a degree of randomness, depending on probability distributions whose parameters can be tuned. The process is repeated until a termination criterion is met.

The design of a GA for a given domain problem requires the specification of the following major elements: (i) a genetic coding of a solution; (ii) a choice of genetic operators and parameters; (iii) a fitness function, to evaluate a solution. These issues are covered in next subsections.

3.1 The Genetic Coding

Let V_n be an input vector at clock cycle n, i.e., $V_n = [v_1, \ldots, v_I]$, where I is the number of input signals of the circuit. A test pattern P is a sequence of consecutive input vectors, i.e., $P = [V_1, \ldots, V_N]$, where N is the number of clock cycles of test pattern P. A test pattern is a chromosome and it is represented by a matrix of size $N \times I$. The following matrix shows the genetic coding of a test pattern:

$$P = \begin{bmatrix} v_{1,1} \cdots v_{1,i} \cdots v_{1,I} \\ \vdots \\ \boxed{v_{n,1} \cdots v_{n,i} \cdots v_{n,I}} \\ \vdots \\ v_{N,1} \cdots v_{N,i} \cdots v_{N,I} \end{bmatrix} \leftarrow gene$$

The n-th row of the matrix represents the gene corresponding to the input vector V_n provided at the n-th clock cycle. The i-th column corresponds to the sequence of values on the i-th input pin.

It is worth noting that the number of genes in chromosomes is not supposed to be constant, since the number of clock cycles can assume a different value for each test pattern.

3.2 The Genetic Operators and Parameters

Crossover is the main genetic operator. It consists in splitting two chromosomes in two or more sub-sequences and obtaining two new chromosomes by exchanging gene sub-sequences between the two original chromosomes. The place where a sub-sequence starts is called a *cut-point*. More specifically, we adopt a *single-point* crossover (Fig. 2) by choosing a random cut-point for each parent and generating the descendants by swapping the following segments, i.e., the segments containing the ending clock cycles. The rationale for this choice is summarized in the following considerations.

With sequential logic the output of a circuit depends on both the current input values and the previous inputs starting from the initial state. Therefore, in order to take advantage of the added benefit of a gene sequence, in terms of number of recognized faults, we should take into account the state of the circuit,

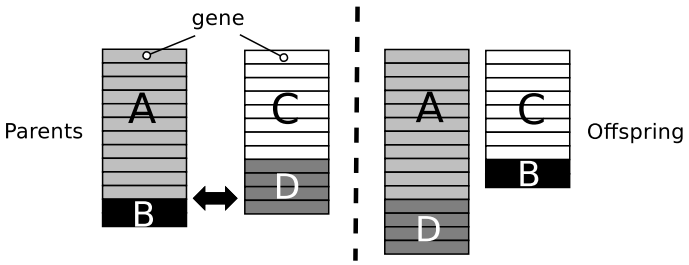


Fig. 2. A scenario for the crossover operator.

which is a result of all previous inputs, i.e., the previous gene sequence. Hence, it is generally more efficient to have a new generation chromosome retain a large fraction of the previous sequence.

In order to achieve this behaviour, we added the following criterion in the crossover operation: Random cut-points are generated via the probability density function of an exponential distribution, i.e., $f(x;\lambda) = \lambda e^{-\lambda x}$, where x is the position of the cut point, i.e., the length of the initial sequence. This distribution implies that a large initial segment is kept unchanged from parent to son. Consequently, the end segments that are swapped are relatively short. The level of exploitation of the previous gene sequences can be adjusted via parameter λ.

Crossover is stochastically activated. The *crossover rate* is the fraction of chromosomes to undergo the crossover operation. A higher crossover rate allows a better exploration of the space of solutions. However, too high a crossover rate causes unpromising regions of the search space to be explored. A typical value is 0.4 [9].

Mutation is an operator that produces a random alteration in a single bit of a gene. Mutation is randomly applied. The *mutation rate*, p_m, is defined in terms of the percentage of new genes in the population that can be introduced. If it is too low, many genes that would have been useful are never discovered, but if it is too high, there will be much random perturbation, the offspring lose their resemblance to the parents, and the GA loses the efficiency in learning from the search history. Typically p_m is about 0.01 [13]. We control the mutation operator by a dynamic p_m which is linearly decreasing, starting with $\overline{p_m}$, down to $\underline{p_m}$ at the final generation.

3.3 The Fitness Function

The fitness function measures the quality of the solution, and is always problem dependent. In our approach fitness has been defined considering both the number of detected faults and the cost of a clock cycle:

$$O(H_j; N_j) = H_j/H_{\text{tot}} - \psi \cdot N_j/N_{\max}$$

where H_j is the number of faults covered by test pattern P_j, H_{tot} is the total number of faults, N_j is the number of clock cycles of test pattern P_j and $N_{\max}$

is the maximum number of clock cycles. Finally, ψ is a cost parameter related to the clock cycle.

We introduced parameter ψ in order to optimize both the fault coverage and the corresponding test pattern length. High values of ψ induce the GA to explore short test patterns at the cost of probably low fault coverages. Conversely, when a low number of clock cycles is not a strict requirement, low values of ψ leave the GA free to explore long test patterns probably obtaining high values of fault coverage.

4 The Test Pattern Generation Tool

The GA is tightly coupled with the simulation-based fault injection tool for FPGAs presented in [3]. In this tool the netlist of a digital circuit is modeled with the *Stochastic Activity Networks* (SAN) [19] formalism using the Möbius [6] modeling and analysis tool. Faults are injected into the model and their propagation is traced to the output pins, using a four-valued logic [3] that enables faulty logical signals to be tagged and recognized without recurring to a comparison with the expected output values.

The GA feeds the fault simulator with the current population of test patterns and then it waits for the fault coverage values produced as output of the simulations. A parser translates the EDIF description of the netlist into the format used by the simulator to instantiate the model of the FPGA-based system, so the tool can automatically interact with the standard design process of an FPGA application. The architecture of the test pattern generation process is shown in Fig. 3.

For each test pattern from the current generation every fault is injected, one at a time and at the beginning of each simulation run. Let *correct system* denote the system in absence of faults. For a given test pattern the simulation process can be summarized in the following steps:

1. A fault is injected in the correct system.
2. The test pattern is applied to the system.
3. The system is executed.
4. If for at least a clock cycle the actual output of the system is different from the expected one, the fault is marked as detected, otherwise it is marked as undetected.
5. If more faults have to be injected, the simulation re-starts from step 1, otherwise it terminates.

At the end of the simulation, the fault coverage of the test pattern is available to the fitness function of the GA.

The GA is an efficient pattern generator thanks to the iterative processing of blocks of test patterns, which sensibly reduces the search space. It can be demonstrated that: (i) selection increasingly chooses test patterns which are better than average; (ii) crossover creates groups of similar patterns avoiding as much as possible to worsen the quality of the patterns resulting from selection;

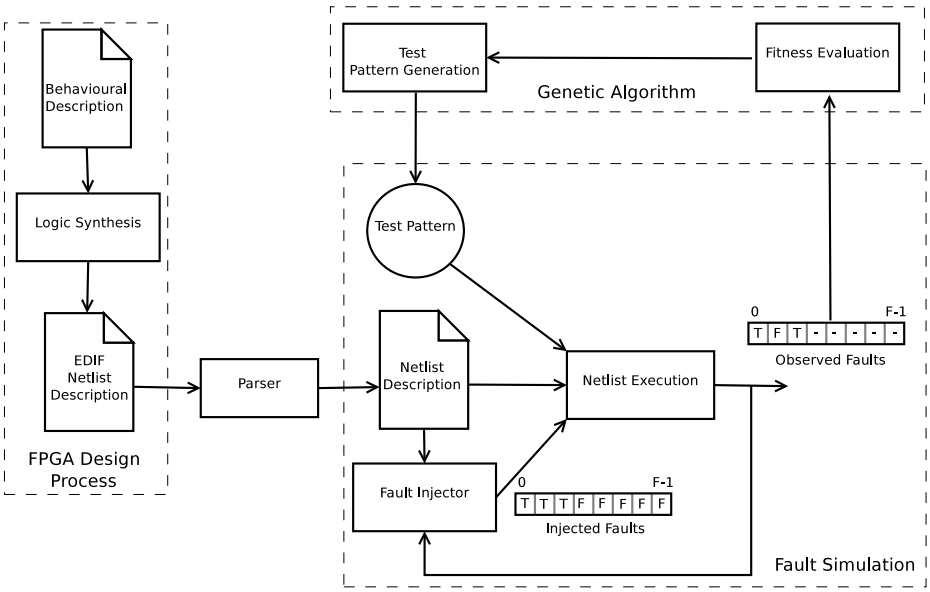


Fig. 3. The Test Pattern Generation process.

and (iii) mutation creates dissimilar patterns without interfering with the result of crossover, especially after a number of generations.

5 Experimental Results

We applied the proposed test pattern generator to some circuits from the ISCAS'85, ISCAS'89 and ITC'99 suites. The GA stops if there is no improvement in the best fitness of the population over a prefixed number of generations, or when the preset maximum number of generations is reached.

To show the character of the optimization process performed by the GA, in Fig. 4 we report, for the ITC'99 b01 circuit, the fault coverage and the fitness function of the best chromosome of each generation versus the number of generations. It may be observed how the GA is able to efficiently increase the fitness function as determined by the trade-off between fault coverage and number of clock cycles.

Results for the considered circuits are shown in Table 1. The first columns specify the circuit name, the corresponding number of VHDL lines (as a rough measure of complexity) and of faults; the *Genetic Algorithm* columns specify the fault coverage (FC) obtained using the proposed GA, the length of the generated test pattern (N) and the computational time per simulated clock cycle required by the GA (T); the Random column specify the fault coverage obtained with a random test pattern of length 10000, while the Random* column specify the

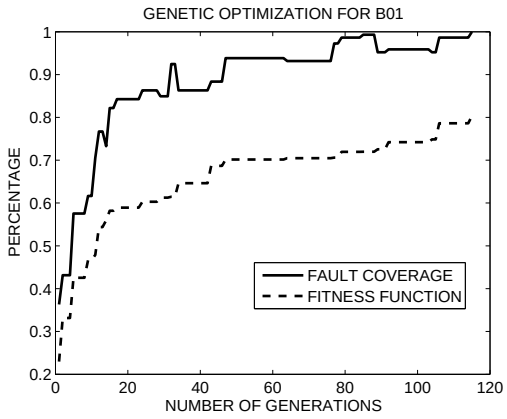


Fig. 4. The optimization process performed by the GA in a trial.

fault coverage obtained with a random test pattern of the same length as the test pattern used by the GA.

The results were obtained on a computer with Intel Core i5 (QuadCore) 2.67 GHz, 256 KB L1 Cache, 1 MB L2 Cache, 8MB L3 Cache, 4 GB RAM.

Table 1. Experimental results.

Circuit name	Characteristics		Genetic Algorithm			Random	Random*
	code lines	faults	FC (%)	N	T (ms)	FC (%)	FC (%)
ISCAS'85 c17	11	39	100.0	17	0.519	100.0	74.3
ITC'99 b02	61	59	89.1	27	0.993	89.8	16.4
ISCAS'89 s27	33	76	83.6	64	1.56	89.0	54.8
ITC'99 b06	112	113	92.9	42	6.25	88.5	23.9
ITC'99 b01	96	146	100.0	52	3.11	95.7	21.2
ISCAS'89 s386	198	689	93.9	4049	448.4	96.8	88.2

Even if for some circuits (e.g., c17 and b06) the GA achieves the same or higher FC than random testing, in other cases random testing achieves a better FC (e.g., circuit s27, with a FC of 89% versus 83.6% for the GA). However, the higher FC with random testing is achieved at a higher cost in terms of test length. The test length with random testing is between 2.47 and 588 times the test length of the GA.

If random testing were to be performed with the number of clock cycles generated by the GA, the FC would be consistently inferior as shown in column Random*.

Finally, we observe that FC reported in the literature for other test pattern generators [5, 17], are generally much higher than the ones shown here, but the different fault model must be taken into account. Those test pattern generators address stuck-at faults, whereas the one discussed here addresses SEUs in any configuration bit. These faults are arguably more difficult to detect than stuck-at faults, and, as observed in Sec. 2, much more numerous, thus our results can hardly be compared with results in the literature.

6 Conclusions and Future Work

We have presented a prototype of an automatic test pattern generation tool for application-dependent testing of SEUs in FPGAs based on a GA. Test patterns generated by the proposed tool can be used for in-service testing of FPGAs. The approach targets SEUs in any configuration bit of the logic resources of the FPGA-based system and this makes our fault model very accurate, but it also makes the system very hard to test. Preliminary results for some circuits from ISCAS'85, ISCAS'89 and ITC'99 benchmarks have been presented.

As future work, faults in the routing resources should be considered. Moreover we intend to design new crossover operators and fitness functions in order to improve both the efficacy and the efficiency of the GA. Another development concerns the exploitation of distributed execution in order to speed up the computations.

Acknowledgements

The authors would like to thank Daniele Lazzarini and Alessandro Rosetti, who implemented the GA in their *Laurea* (Bachelor's Degree) thesis.

References

1. R.C. Baumann. Radiation-induced Soft Errors in Advanced Semiconductor Technologies. *IEEE Transactions on Device and Materials Reliability*, 5(3):305 – 316, September 2005.
2. C. Bernardeschi, L. Cassano, and A. Domenici. Failure Probability and Fault Observability of SRAM-FPGA Systems. In *International Conference on Field Programmable Logic and Applications (FPL2011)*, pages 385 –388, sept. 2011.
3. C. Bernardeschi, L. Cassano, A. Domenici, G. Gennaro, and M. Pasquariello. Simulated Injection of Radiation-Induced Logic Faults in FPGAs. In *Proceedings of the 3rd International Conference on Advances in System Testing and Validation Lifecycle (VALID 2011)*, 2011.
4. F. Corno, P. Prinetto, M. Rebaudengo, and M. Sonza Reorda. Gatto: a genetic algorithm for automatic test pattern generation for large synchronous sequential circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 15(8):991 –1000, aug 1996.
5. F. Corno, M. Sonza Reorda, and G. Squillero. RT-Level ITC'99 Benchmarks and First ATPG Results. *IEEE Des. Test*, 17:44–53, July 2000.

6. Daniel D. Deavours, Graham Clark, Tod Courtney, David Daly, Salem Derisavi, Jay M. Doyle, William H. Sanders, and Patrick G. Webster. The Möbius framework and its implementation. *IEEE Trans. Softw. Eng.*, 28(10):956–969, 2002.
7. European Committee for Electrotechnical Standardization (CENELEC). EN 50129: Railway applications - Communications, signaling and processing systems - Safety related electronic systems for signaling, February 2003.
8. International Organization for Standardization (ISO). 26262-5: Road vehicles - Functional safety - Part 5. Product development: hardware level, December 2009. Draft.
9. M. Gen and R. Cheng. *Genetic Algorithms and Engineering Design*. John Wiley & Sons., 1997.
10. P. Graham, M. Caffrey, J. Zimmerman, D. E. Johnson, P. Sundararajan, and C. Patterson. Consequences and Categories of SRAM FPGA Configuration SEUs. In *Proceedings of the 6th Military and Aerospace Applications of Programmable Logic Devices (MAPLD'03)*, September 2003.
11. W.K. Huang, F.J. Meyer, N. Park, and F. Lombardi. Testing Memory Modules in SRAM-based Configurable FPGAs. In *Proceedings of the International Workshop on Memory Technology, Design and Testing*, pages 79 –86, aug 1997.
12. International Atomic Energy Agency (IAEA). NS-G-1.3: Instrumentation and Control Systems Important to Safety in Nuclear Power Plants, 2002. IAEA Safety Standards Series.
13. Z. Michalewicz. *Genetic Algorithms Plus Data Structures Equals Evolution Programs*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2nd edition, 1994.
14. M. Rebaudengo, M.S. Reorda, and M. Violante. A new functional fault model for FPGA application-oriented testing. In *Proceedings of the 17th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT 2002)*, pages 372 – 380, 2002.
15. M. Renovell, J.M. Portal, P. Faure, J. Figueras, and Y. Zorian. Analyzing the Test Generation Problem for an Application-Oriented Test of FPGAs. In *Proceedings of the IEEE European Test Workshop*, pages 75 –80, 2000.
16. M. Renovell, J.M. Portal, J. Figuras, and Y. Zorian. Minimizing the Number of Test Configurations for Different FPGA Families. In *Proceedings of the Eighth Asian Test Symposium (ATS '99)*, pages 363 –368, 1999.
17. M. Rozkovec, J. Jenicek, and O. Novak. Application Dependent FPGA Testing Method. In *Proceedings of the 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools (DSD10)*, pages 525 –530, sept. 2010.
18. E.M. Rudnick, J.H. Patel, G.S. Greenstein, and T.M. Niermann. A Genetic Algorithm Framework for Test Generation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 16(9):1034 –1044, sep 1997.
19. W. Sanders and J. Meyer. Stochastic activity networks: formal definitions and concepts. In E. Brinksma, H. Hermanns, and J.P. Katoen, editors, *Lectures on Formal Methods and PerformanceAnalysis*, volume 2090 of *Lecture Notes in Computer Science*, pages 315–343. Springer Berlin / Heidelberg, 2001.
20. Yu.A. Skobtsov and V.Yu. Skobtsov. Evolutionary approach to test generation of sequential digital circuits with multiple observation time strategy. In *Proceedings of the East-West Design Test Symposium (EWDTS10)*, pages 286 –291, sept. 2010.
21. C. Stroud, S. Wijesuriya, C. Hamilton, and M. Abramovici. Built-in Self-Test of FPGA Interconnect. In *Proceedings of the International Test Conference*, pages 404 –411, oct 1998.
22. M. Tahoori. Application-Dependent Testing of FPGAs. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(9):1024 –1033, 2006.