GABES: A genetic algorithm based environment for SEU testing in SRAM-FPGAs ☆

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

Department of Information Engineering, University of Pisa, Italy

A B S T R A C T

Testing of FPGAs is gaining more and more interest because of the application of FPGA devices in many safety-critical systems. We propose GABES, a tool for the generation of test patterns for application-dependent testing of SEUs in SRAM-FPGAs, based on a genetic algorithm. Test patterns are generated and selected by the algorithm according to their fault coverage: Faults are injected in a simulated model of the circuit, the model is executed for each test pattern and the respective fault coverage is computed. We focus on SEUs in configuration bits affecting logic resources of the FPGA. This makes our fault model much more accurate than the classical stuck-at model. Results from the application of the tool to some circuits from the ITC'99 benchmarks are reported. These results suggest that this approach may be effective in the inspection of safety-critical components of control systems implemented on FPGAs.

Keywords:
Genetic algorithm
Test pattern generation
SEUs
SRAM-FPGAs

1. Introduction

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) [2].

In the last years FPGAs have increasingly been employed in safety-related and safety-critical applications such as railway signaling [3], radar systems for automotive applications [4] and wireless sensor networks for aerospace [5].

The industrial use of electronic devices in safety-critical systems is regulated by application-related safety standards that impose strict safety requirements on the system. In particular, safety standards such as ISO 26262-5 [6], CENELEC 50129 [7], and IAEA NS-G-1.3 [8], require in-service testing activities for safety-related systems. It is therefore vital that the tests be able to detect the largest number of faults that may occur in the system.

Automated test generation aims at finding input values (structured in test vectors, test patterns, or sets of test patterns) that can detect a large number of faults, while minimizing testing time.

Two main families of test methods for FPGA circuits exist: *application-independent* and *application-dependent*. Application-independent methods [9–11] aim at detecting structural defects due to the manufacturing process in the whole FPGA chip. Conversely, application-dependent methods [12,13] address only the resources of the FPGA chip actually used by the implemented system.

A common way to obtain an efficient set of test patterns is to generate many patterns randomly, computing their fault coverage by simulation, and selecting a set that covers all the detectable faults. This process, however, is time-consuming and may not yield optimal test sets with respect to the conflicting requirements of high fault coverage and short testing time. Techniques based on *evolutionary search* may then be used to improve the selection process [14–17].

Many works addressing the problem of automatic test pattern generation (ATPG) for digital circuits have been published [18], but very few of these works specifically address FPGAs. Test methods devised for ASIC circuits could be effective when used for testing structural defects in the FPGA chip, but they are not satisfactory when used for testing SEUs in the configuration memory of FPGAs [19]. In particular, it has been demonstrated [20] 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 considers permanent faults at the input and output terminals of the logical components. More accurate fault models, keeping into account faults in the configuration bits of the FPGA chip, should be considered.

In this work, the model proposed in [20] has been adopted. In this model, an SEU in the configuration bit of a component causes a faulty output of the same component if and only if the input values of the component are exactly those associated with the faulty configuration bit. Such faults are more difficult to detect than

☆ A preliminary version of this paper appeared as *Application of a Genetic Algorithm for Testing SEUs in SRAM-FPGA Systems* in the Proceedings of the 6th HiPEAC Workshop on Reconfigurable Computing (WRC 2012), 2012.

* Corresponding author. Tel.: +39 0502217674.

E-mail addresses: c.bernardeschi@ing.unipi.it (C. Bernardeschi), luca.cassano@ing.unipi.it (L. Cassano), m.cimino@ing.unipi.it (M.G.C.A. Cimino), a.domenici@ing.unipi.it (A. Domenici).

stuck-at faults, since they are excited or not, depending on the input of the component.

Genetic algorithms (GA) have been shown to be effective as test pattern generation engines compared to deterministic methods [21,22].

In this work, we propose GABES (Genetic Algorithm Based Environment for SEU testing), a tool for automatic test pattern generation based on a GA for application-dependent testing of SEUs in SRAM-FPGAs, that takes into account SEUs in configuration bits of the FPGA. Modern FPGAs are very large and complex chips, and thus it is unfeasible for a test generation method to target every possible fault [19]. We focus on SEUs affecting logic components of the FPGA, leaving out SEUs affecting the routing structure, which will be object of further work. Dealing with the routing structure separately from the logic components is motivated by the complexity of the former [19] and by the fine granularity of the adopted fault model. However, the GA does not need any knowledge of the circuit internals and is therefore agnostic with respect to the nature and location of faults. The presented methodology can then be applied also to the testing of routing faults.

The proposed GA uses the simulation-based fault injection tool for FPGAs presented in [23] 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 [24,25].

A design choice in the development of the tool was to use a relatively lightweight GA. Other proposals in the literature have different approaches, where the GA has access to details of the circuit structure and functionality [14], whereas in the present work the GA relies only on the externally observable behavior of the simulated circuit. This choice is motivated by the purpose of reducing the design complexity of the test pattern generator. In particular, this choice leads to a simpler and faster evaluation of the fitness function, and to a modular TP generation process, where knowledge of circuit internals is needed only by the simulator. Recourse to heuristics depending on the circuit's internals, instead, could produce better TPs, but it would make the process more cumbersome and less flexible.

The main goal of this work is producing efficient sets of test patterns for in-service testing, that can be optimized with respect either to fault coverage or to test speed, according to the specific application requirements. Another goal is optimizing the test pattern generation itself: Even if the off-service process of test pattern generation is not subject to the stringent constraints of in-service testing, excessive computation times can make the method impractical. We note that our methodology is oriented to application-dependent testing and does not address application-independent testing, due to the abstraction level of the adopted circuit and fault models.

The remainder of the paper is organized as follows: Section 2 is a brief survey of works addressing test pattern generation in digital circuits; in Section 3 the considered fault model is presented, fault simulation is briefly discussed and basic concepts of evolutionary approaches to problem solving are introduced; in Section 4 the main characteristics of the proposed GA are discussed; Section 5 briefly presents the structure of the test pattern generation tool; Section 6 presents the results of the proposed algorithm for some circuits from the ITC'99 benchmarks; Section 7 concludes the paper.

2. Related work

The main approaches to counteracting the effects of SEUs are analysis and fault tolerance techniques. Analysis activities include accelerated ground radiation testing [26], fault injection boards [27], analytical computation [28], and fault simulation [29]. Fault tolerance [30] is achieved either by design e.g, time, information or hardware redundancy, or by use of robust technologies e.g, radiation hardening [31] and error detection and correcting codes [32]. Analysis-based approaches are a valid complement to approaches relying on robust design or technology.

Many approaches to automatic test generation for digital circuits [18] are found in the literature. A broad classification can be made between *deterministic* and *random* test generation methods. Deterministic methods are based on algorithms, such as the D-algorithm [33], PODEM [34], or FAN [35], that rely on knowledge of the circuit structure to compute sets of test vectors that can detect all possible stuck-at faults. These techniques are generally able to generate test patterns with a high fault coverage and an optimized length, but they suffer from long execution times. Random methods [36] produce test vectors as pseudo-randomly generated n-tuples of input values, thus requiring no knowledge of circuit structure. Random methods generate test vectors more quickly than deterministic methods, but need a large number of vectors to ensure a high probability of detecting all faults. Newer pseudo-random techniques use re-seeding and bit changing to improve fault coverage [37,38]. Some algorithms, such as RAPS [39] and SMART [40], combine random techniques with structural information in order to improve the efficiency of randomly generated test sets.

Another way to improve the quality of randomly generated test sets is using *coverage-directed* generation [41,42]. This is an iterative and evolutionary approach, where at each step the fault coverage of a group of tests is evaluated by simulation, and at the next step the group is transformed in order to improve fault coverage and other desirable properties. Many techniques and criteria can be used to generate new tests at each step. In particular, *genetic algorithms* [43–45] have proven to be effective.

Early applications of genetic algorithms to test pattern generation were presented by Saab et al. [46], Rudnick et al. [47], and Corno et al. [14]. In the last twenty years, genetic algorithms have been proposed for many tasks in validation and testing of digital circuits. Genetic algorithms have been used for test pattern generation addressing hardware defects in digital circuits [48,49], for test program generation addressing microprocessor defects [50,51] and microprocessor functional validation [52].

In the area of FPGA testing, two families of methods may be distinguished: *application-independent* and *application-dependent* methods. Application-independent methods, such as those reported by Huang et al. [9], Renovell et al. [10], and Stroud et al. [11], aim at detecting structural defects due to the manufacturing process of the chip. These techniques are mainly performed by the chip manufacturer, and thus they are also known as *manufacturer-oriented* techniques. 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 [12,13] address only those resources of the FPGA chip actually used by the implemented system. Since these techniques are applied by the user after the system design has been defined, they are also known as *user-oriented*. The basic idea behind this family of techniques is that very often an FPGA-based system uses only a subset of the resources provided by the FPGA chip. Therefore, demonstrating that the resources used by the implemented system are fault-free is sufficient to guarantee the correct operation of the system itself. Application-dependent methods have been proposed for in-service testing of both structural defects [12,13] and SEUs [23].

3. Background

This section provides some background information about the effects of SEUs in the configuration memory of an FPGA, about SEU simulation and about genetic algorithms.

3.1. The fault model

An FPGA [53] 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 functionality 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.

This work adopts a circuit simulation model corresponding to the detail level of the netlist representation of FPGAs, produced in the synthesis phase before the place and route phase. At this level, the elements visible in the model are I/O buffers, LUTs, flip-flops, and multiplexers. Since the latter two are not configurable, only SEUs in the configuration memory of I/O buffers and LUTs need be considered.

In the stuck-at fault model, an SEU in the configuration memory of a component causes the output of the component to be stuck at a given value, thus the fault is always active. In our approach, instead, SEUs are modeled at a finer detail, since, in this fault model, an 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 to the LUT is the one associated with the faulty configuration bit. Fig. 1(a) shows an SEU causing a bit flip in the configuration bit associated with input (0000). 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 (0000) 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.

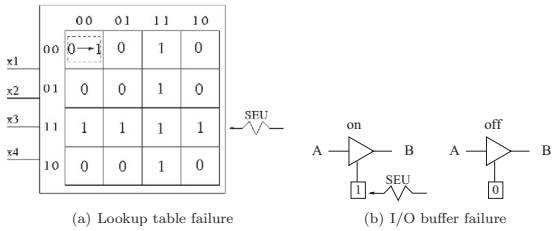
An SEU in the configuration bit of a buffer causes an undesired connection or disconnection between two wires, as shown in Fig. 1(b).

Fig. 2 shows LUT L from Fig. 1(a) in the context of its device D, where C represents the rest of the device, and the *i*'s and *z*'s represent the n primary inputs and the m primary outputs, respectively.

A fault in L may be detected by applying a test pattern, i.e., a sequence of n-tuples of values (test vectors) to the primary inputs. A test vector applied to D *activates* a fault if network C applies to L the input configuration that selects the faulty configuration bit, in this case (0000). This may be possible or not, depending on the structure and current state of C: For example, if C is such that two input pins of L (say, x_1 and x_2) will always have complementary values, that fault is unexcitable.

If a fault is activated, the resulting erroneous value produced by the component may *propagate* to the primary outputs. Again, error propagation depends on the structure and state of C.

A fault is *detected* by a given test pattern if the test pattern activates the fault and propagates an erroneous value to one or more primary outputs.



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

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

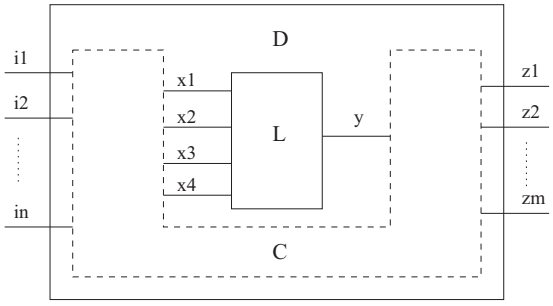


Fig. 2. A LUT within a device.

3.2. Fault simulation

Simulation enables designers to study a system's behavior in presence of faults by exercising a software model of the system. In a simulation context, fault injection consists in modeling the possible faults, and the type and number of faults that can be injected is closely related to the level of abstraction of the system's model.

Fault simulation may be done for different purposes, such as finding efficient tests for production testing or in-service testing, or assessing the system's robustness, or its testability [24]. This work is focused on generating sets of tests that strike a balance between a high fault coverage and a short test execution time.

With reference to Fig. 2, LUT fault simulation consists in the execution of a series of simulation runs. At each run, a new test pattern is repeatedly applied to the system's model, once for each possible fault. Each fault is then tested independently, under the single-fault hypothesis. More precisely, for each test pattern the following steps take place:

1. A bit flip in a configuration bit k of L is injected in the model.
2. The test pattern is applied to the primary inputs.
3. If the input configuration $(x_1, \ldots, x_4)$ to L selects configuration bit k, an error is found at node y.
4. If for at least one clock cycle one or more primary outputs take an erroneous value, a failure has occurred and the fault is marked as detected.
5. If more faults are still to be injected, the run re-starts from step 1 with the injection of a new fault, otherwise the run terminates.

The same steps are executed when considering SEUs in the configuration bits associated with I/O buffers, with the only difference that the output of a faulty I/O buffer remains stuck at its last value before the fault occurred.

At the end of the run, the fault coverage of the test pattern is computed as the ratio of detected faults to injected faults.

3.3. Evolutionary approaches

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 [44].

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.

More precisely, for a given optimization problem, an *initialization* process provides a set of randomly generated approximated solutions. Each solution is then *evaluated*, using an appropriate measure of fitness. If the *termination* criteria are satisfied, a solution is then *elected* as (sub)optimal for the problem. If not, each solution is encoded as a chromosome. The chromosomes evolve through successive generations, i.e., iterations of the GA. During each generation, a set of new chromosomes, called an *offspring*, is formed by: (i) *selection* of a *mating pool*, i.e., a quota of parent chromosomes from the current population, according to the fitness values; (ii) combination of pairs of parents via the *crossover* genetic operator; (iii) modification of offspring chromosomes via the *mutation* genetic operator. The new chromosomes are then *decoded* in terms of domain solutions. Finally, a new generation is formed by *reinserting*, according to their fitness, some of the parents and offspring, and rejecting the remaining individuals so as to keep the population size constant.

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.

4. The genetic algorithm

The tool presented in this work produces a *test set* (TS), i.e., a set of test patterns, each one selected from the population generated at some step of a GA. More precisely, the GA maintains a *Dynamic Global Record Table* (DGRT) [54] containing a list of test patterns with the respective sets of detected faults. At each generation, the fitness of each individual from the population is evaluated. Then the individuals are examined in descending order of fitness and an individual is inserted in the DGRT if it detects faults that have not yet been found by previously inserted individuals. The construction of the DGRT is completed when its entries cover all faults (or a preset number of iterations has been reached), and the test patterns in the table are the final TS.

The information in the DGRT is also used to compute the fitness function, as explained in Section 4.4.

4.1. The genetic coding

Single test patterns are considered as individuals (or chromosomes) in the GA. Their genetic coding, described below, is a matrix of logic values.

Let V_i be an input vector at clock cycle i, i.e., $V_i = [v_1, \ldots, v_n]$, where n is the number of input signals of the circuit, and the v's are the respective values. A test pattern (TP) is a sequence $[V_1, \ldots, V_l]$ of consecutive input vectors, where l is the number of clock cycles (or *length*) of the test pattern. Therefore, a test pattern is represented by a matrix of size $l \times n$, as shown in Fig. 3.

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

It may be noted that the number of genes in chromosomes is not assumed to be constant, since the number of clock cycles can take a different value for each test pattern.

4.2. 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. 4) by choosing a non-uniform cut-point for each parent and generating the descendants by swapping 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, 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 behavior, 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 distance of the cut point from the end of the sequence. This distribution implies that a large initial segment is kept unchanged from parent to child. 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.

The fraction of chromosomes to undergo the crossover operation is the *crossover rate* (p_c). The crossover operator is applied with a probability p_c on the selected pair of individuals. When the operator is not applied, the offspring is a pair of identical copies, or *clones*, of the parents.

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. Typical values are in the order of 10^{-1} [44].

Mutation is an operator that produces a random alteration in a single bit of a gene. Mutation is randomly applied. The *mutation rate*, p_μ, is defined as the probability that an arbitrary bit of an arbitrary gene is complemented. 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. Typical values of p_μ are in the order of 10^{-2} [43]. We control the mutation operator by a dynamic p_μ which decreases linearly between an initial value $\overline{p}_\mu$, and a value $\underline{p}_\mu$ at the final generation.

With a linearly decreasing p_μ, the early generations have a high probability of mutation and solutions are spread all over the solutions space, so that most of them have a chance to be tried. Later generations have a lower mutation probability, so that the search

$$\begin{bmatrix} v_{1,1} \cdots v_{1,j} \cdots v_{1,n} \\ \vdots \\ \boxed{v_{i,1} \cdots v_{i,j} \cdots v_{i,n}} \\ \vdots \\ v_{l,1} \cdots v_{l,j} \cdots v_{l,n} \end{bmatrix} \leftarrow i\text{-th gene}$$

Fig. 3. Genetic coding of a test pattern.

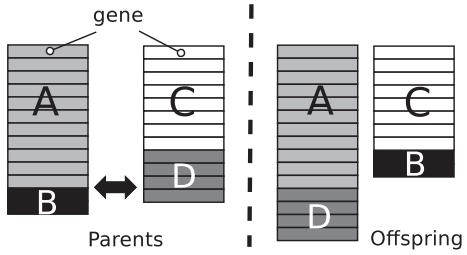


Fig. 4. A scenario for the crossover operator.

is focused on the regions of the solution space where fitter individuals are found.

4.3. Selection method

A selection operator chooses a subset of chromosomes from the current population. Various stochastic selection techniques are available. In this work the *roulette wheel* method [55] is used. With this method, an individual is selected with a probability that is directly proportional to its fitness. Each individual is mapped to an arc of a circle whose length equals the individual's fitness. The circumference is then equal to the sum of the fitnesses. Selection is made by choosing a random number with a uniform distribution between 0 and the circumference. The selected individual is the one mapped to the arc containing the chosen point. This ensures that better fit individuals have a greater probability of being selected, however all individuals have a chance.

4.4. The fitness function

The fitness function measures the quality of the solution, and is always problem dependent. In our approach, fitness takes into account the fault coverage achieved by each TP, its length, and its effectiveness in finding *hard* faults.

The fitness function adopted in this work relies on a DGRT to evaluate each TP with respect to the performance of previously generated TPs.

The fitness function of a test pattern i is defined in terms of a value c_i that we call the *relative efficiency* of the TP:

$$c_i = \sum_{j=1}^{n_i} (\xi_{ij} + 1)^{-k},$$

where n_i is the number of faults detected by the i-th test pattern; ξ_{ij} is the number of test patterns, generated before the i-th one, that detect fault j; k is is a configurable parameter of the algorithm, ranging in $[0, 1]$.

The fitness function is then

$$f(i) = \frac{c_i}{N} - M\frac{l_i}{L},$$

where N is the number of injected faults and M is a configurable parameter of the algorithm, ranging in $[0, 1]$, which represents the relative cost per clock cycle. For M equal to zero, the optimization process tends towards a maximum coverage. For increasing values of M, the fitness function penalizes also large test patterns. Parameter l_i is the length of the i-th test pattern; L is the maximum length of the test patterns. This parameter is chosen heuristically, depending on the size and complexity of the circuit.

Table 1 summarizes the parameters of the fitness function, together with those occurring in the TP generation algorithm (Section 4.5). The table also reports the values assigned to the parameters in the experiments discussed in Section 6.

Table 1
Parameters.

Parameter	Description	Value
λ	Rate parameter for cut-point distribution	1
p_c	Crossover rate	0.8
$\overline{p}_\mu$	Maximum mutation rate	0.15
$\underline{p}_\mu$	Minimum mutation rate	0.05
N	Number of possible faults	–
M	Penalty coefficient for TP length	0.5
L	Maximum length for TPs	10000
k	Penalty exponent for easy-to-detect faults	0.75
$s_{\max}$	Maximum number of iterations	2000
S	TP population size	200
m	Number of iterations considered for stall condition	20
$\bar{n}$	Threshold for acceptance into DGRT	20
m_d	Number of iterations considered for $\bar{n}$ adaptation	20

The fitness function increases with an individual's relative efficiency c_i. This value increases with the number of faults detected by individual i, but the weight of each detected fault j decreases with the number ξ_{ij} of other individuals that have been shown to detect the fault before i. The number ξ_{ij} is obtained from the DGRT and indicates how easily a fault can be detected (statistically, easy faults are detected earlier and more often). In this way, individuals that detect harder-to-find faults are rewarded. With higher values of parameter k, easy faults detected by a test pattern add a smaller contribution to its fitness.

4.5. Producing the test set

The final TS is obtained by an overall algorithm that iteratively evaluates a population of test patterns, inserts the best ones in the DGRT, and calls the GA proper to improve the population. The GA, in turn, uses the DGRT to compute the fitness function, as shown in the previous subsection. This is described more formally in Algorithm 1, where s identifies the iterations of the algorithm (up to a limit of $s_{\max}$ iterations), D is the DGRT, and P_s is the test pattern population at iteration s. The size of the population is S, and N is the number of possible faults. Parameter S is chosen so as to guarantee adequate diversity among individuals while limiting the computational cost of fitness evaluation. Predicate improve(m) is false when a *stall* condition occurs, i.e., when no improvement in the fitness of the best individual of each generation is achieved over the last m iterations.

Algorithm 1. The overall algorithm.

```
s ← 0; D ← ∅; D′ ← ∅
P_0 ← (S randomly generated test patterns)
while coverage(D) < N ∧ improve(m) ∧ s < s_max do
  for i = 1 to S do
    for j = 1 to N do
      if detects(i, j) then
        detected(i) ← detected(i) ∪ {j}
      end if
    end for
    if finds(i, n̄, D) then
      D ← D ∪ new(i, n̄, D)
    end if
  end for
  s ← s + 1; P_s ← ga(P_{s−1}, D)
end while
D′ ← compact(D)
return testset(D′)
```

For ease of notation, we assume that test patterns and faults are identified by natural numbers. The DGRT is represented as a set of pairs (i,j), such that test pattern i detects fault j. Function coverage(D) is the number of faults detected by the test patterns recorded in the DGRT, and detects(i,j) is true if and only if test pattern i detects fault j. The set of faults detected(i) found by test pattern i is updated in the course of the simulation. Predicate finds$(i,\bar{n},D)$ is true if and only if test pattern i detects a set of at least $\bar{n}$ faults not yet recorded in the DGRT, and new$(i,\bar{n},D)$ returns the pairs $(i,j_1),\ldots,(i,j_{\bar{n}})$ such that test pattern i detects a fault in that set. The value of $\bar{n}$ is set initially at 20. When the number of new detected faults drops below the current value of $\bar{n}$ for m_d consecutive iterations, the value is decreased gradually. More precisely, $\bar{n}$ is set equal to the highest number of new faults found by a single individual in the current generation.

The GA ga(P_{s-1},D) produces the new generation P_s from the previous one, using the DGRT to compute the fitness. On exit from the outermost loop, function compact(D) produces a new DGRT by removing individuals, or terminal segments thereof, whose faults are covered by other ones. Finally, testset(D) returns the test patterns contained in the DGRT.

The algorithm stops when one of the following conditions holds: (i) total fault coverage is achieved, or (ii) a stall condition is met, or (iii) the maximum allowed number of iterations $s_{\max}$ is reached.

Algorithm 2. The genetic algorithm.

```
P_M ← ∅; A ← ∅; B ← ∅
for i = 1 to Q do
  x ← select(P); P_M ← P_M ∪ {x}
end for
for i = 1 to Q/2 do
  (x,y) ← pair(P_M);(x',y') ← crossover(x,y,λ)
  x'' ← mutate(x',p_μ);y'' ← mutate(y',p_μ)
A ← A ∪ {x'',y''}
end for
for i = 1 to S − Q do
  x ← select(P,p_c);
  B ← B ∪ {x}
end for
P ← A ∪ B
return P
```

In Algorithm 2, P is the current population, P_M is the mating pool, A is the offspring, i.e., the set of individuals resulting from crossover and mutation, B is the set of individuals passed unchanged to the next generation, and Q is the size of the mating pool (Section 3.3).

Function select(P) returns an individual from P, selected with the roulette wheel method, and function pair(P_M) returns a pair of parents from P_M, selected with the roulette wheel method. Function crossover(x,y,λ) returns the offspring of a pair of parents, with λ as the *level of exploitation* parameter for cut point selection (Section 4.2). Mutation is then applied to the selected parents with probability p_μ, and the mutated individuals are added to set A.

Finally, a set B is built, with cardinality $S-Q$, with individuals drawn from the population P passed to the algorithm. The new generation is then obtained by replacing P by the union of A and B.

It may be observed that all sets used in the algorithm may contain pairs of identical individuals, due to the random character of the various operators. However, each individual is identifiable even when it is structurally identical to another one, therefore all sets are proper sets (not multisets). As a consequence, the cardinality of P is a constant.

4.6. The parameter tuning process

The parameter values shown in Table 1 have been set according to a generic optimization strategy, for a set of circuits from the ITC'99 benchmark [56], often used in the field (Section 6). In this subsection, we will summarize the methodology used for parameter tuning.

In evolutionary techniques, a number of application constraints narrow down the choice of parameter values [57]. Such constraints are, in our case, the relative cost per clock cycle, the maximum length allowed for a test pattern, the maximum time allowed for generating a test pattern per circuit (related to the maximum number of generations), the available computation resources for test pattern generation (related to the population size), and so on. Moreover, parameters that cannot be chosen from application constraints can be tuned by using sensitivity analysis. Sensitivity is informally defined as the effect of uncertainty in the parameter on the final results [58].

For the purposes of this paper, application dependent parameters (such as $M, L, s_{\max}, S$) have been set to some prototypical value. Let us consider the parameters intrinsically related to the genetic algorithm. When the parameter sensitivity is low, as for p_c and λ, its value has been set to values commonly adopted in the literature. When the parameter sensitivity is higher, some adaptation technique has been introduced, meaning that the parameter value is dynamically established with the support of a semi-automatic tuning process. Adaptation helps reducing the sensitivity, because it can compensate deviations of a parameter's initial value, chosen within an initial range, from its optimal value.

For example, the value of the mutation rate varies adaptively within a range whose extremes, $\bar{p}_\mu$ and $\underline{p}_\mu$, have been set to values commonly adopted in the literature.

Table 2 shows examples of how variations of two GA parameter affect TP length (len) and TP generation length, in number of generations (gen) and time (in minutes), for a 100% coverage.

Twenty-four simulations of circuits b01, b02, and b06 (Section 6) have been grouped in six sessions of four trials. In each session, only one parameter varies while the other one is fixed. The best performance of a session in terms of TP length and completion time is emphasized with a rectangular box. The best performance over the whole set of twenty-four simulations is further emphasized with boldface style.

It may be observed that: (i) the TP length does not increase for increasing $\bar{n}$, thanks to the adaptation mechanism; (ii) a fast convergence can be guaranteed with a low $\bar{n}$; (iii) however, when $\bar{n}$ is very low, the TP length sensibly increases, because the adaptation mechanism is not able to increase $\bar{n}$; (iv) lower values of m_d allow a better performance in time.

The parameter values adopted for the experiences reported in Section 6 can be applied to benchmarks of different size and complexity, given the intrinsic adaptivity of the GAs. Obviously a finer tuning can be made, relying on application constraints and on sensitivity analysis [58].

With regard to genetic operators, many alternatives are available in the literature. Most of them are used for very specific purposes, unrelated to the aims of this paper. Two general-purpose selection operators, namely, the roulette wheel and rank selection operators [59] has been considered. The former leads to a better exploitation of previous useful mutations, while the latter leads to a wider-range exploration of the solution space. More specifically, rank selection prevents too quick a convergence and differs from roulette wheel selection in terms of selection pressure. Rank selection overcomes problems like stagnation or premature convergence [59]. The GABES choice of using the roulette wheel is due to the considerable amount of exploration already performed by the crossover and mutation operators.

Table 2
Genetic parameters and related performance (boldface highlights best performance values).

		b01			b02			b06		
$\bar{n}$	m_d	gen	len	time	gen	len	time	gen	len	time
5	**5**	17	131	3.17	6	57	**0.22**	9	101	2.98
5	**25**	14	145	**2.27**	6	77	0.25	10	112	3.63
5	35	18	126	2.60	6	73	0.23	6	92	2.12
5	**50**	18	140	3.27	14	76	0.62	5	96	**1.73**
10	5	14	119	2.52	11	106	0.48	9	96	3.43
10	25	27	101	5.18	29	52	1.53	28	99	13.13
10	35	46	141	10.07	12	76	0.55	9	88	3.38
10	50	45	127	10.25	19	67	0.95	18	122	7.38
15	5	16	112	3.10	14	57	0.60	8	72	2.83
15	25	41	103	9.33	21	52	1.08	14	102	6.12
15	35	43	141	10.22	18	61	0.80	38	87	20.00
15	50	64	120	17.38	23	95	1.10	13	108	5.35
20	5	20	147	4.07	11	54	0.47	11	69	4.23
20	25	48	117	11.13	14	81	0.63	28	97	13.37
20	35	44	143	10.30	12	78	0.53	17	113	7.37
20	**50**	48	119	13.20	7	49	0.30	42	**64**	21.50
30	5	24	164	5.10	15	60	0.62	12	69	4.38
30	25	27	109	6.28	29	70	1.53	29	86	13.95
30	**35**	52	107	14.00	27	**39**	1.45	23	96	10.47
30	50	23	105	5.12	19	45	0.88	48	85	25.22
5	20	19	101	3.27	16	70	0.78	12	96	4.48
15	20	30	112	5.47	19	73	0.87	13	69	5.38
20	**20**	35	**94**	7.73	22	61	1.10	22	124	10.42
30	20	32	177	7.57	24	63	1.27	27	94	12.70

5. The test pattern generation tool

The GA discussed above is coupled with the simulation-based fault injection tool for FPGAs presented in [23]. In this tool, the netlist of a digital circuit is modeled with the *Stochastic Activity Networks* (SAN) [60] formalism using the Möbius [61] modeling and analysis tool. Faults are injected into the model and their propagation is traced to the output pins, using a four-valued logic [23] that enables faulty logical signals to be tagged and recognized without recurring to a comparison with the expected output values.

The test pattern generation process is shown in Fig. 5. In the figure, the block labeled "FPGA Design Process" is performed by an external tool that produces a netlist described in the EDIF language [62]. This description is parsed by a tool developed alongside the simulator, which extracts information on topology and LUT functions, and encodes it into a simpler textual format used by the simulator. In this way, GABES can seamlessly interact with the standard design process of an FPGA application.

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. These values are then used to update the DGRT and compute the fitness functions of the test patterns, leading to the next generation of the GA.

This GA is an efficient pattern generator thanks to the iterative processing of blocks of test patterns, which appreciably reduces the search space. It may be noted that its genetic operators have the following properties: (i) At each generation, selection chooses test patterns that are better than average; (ii) crossover creates groups of similar patterns to avoid worsening the quality of the selected patterns; and (iii) mutation creates dissimilar patterns without interfering with the result of crossover, especially in the later generations.

6. Experimental results

The GABES test pattern generator has been applied to some circuits from the ITC'99 suite [56]. The VHDL code of the circuits was synthesized for the Virtex-6 target device using the Xilinx ISE tool [63]. Only the b13 circuit has been synthesized for the Virtex-4 family, due to simulation time constraints. We note that this limitation is not due to the GA but to the simulator, which is still a prototype. This simulator is currently the only one capable to model faults at the desired level of detail.

The characteristics of the netlists, in terms of the number of LUTs, flip-flops (FFs), multiplexers (MUXs) and input and output buffers (IBufs and OBufs), are summarized in Table 3. The table also shows the number of possible faults (Faults) and of excitable faults (Ex), calculated with the SEU-X tool [64]. The function of the circuits, as reported in Corno *et al.* [56], is shown in Table 4. This selection includes typical applications of embedded systems. The values of the parameters for the experiments are shown in Table 1 of Section 4.4.

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 3
Circuit characteristics.

Circuit	LUTs	FFs	MUXs	IBufs	OBufs	Faults	Ex
b01	15	10	1	3	2	141	141
b02	4	4	0	2	1	55	49
b03	90	35	1	5	4	687	269
b06	9	8	0	3	6	153	133
b08	47	21	1	10	4	970	546
b09	47	29	2	2	1	1067	347
b10	55	24	0	12	6	702	348
b13	95	75	15	11	10	1094	485

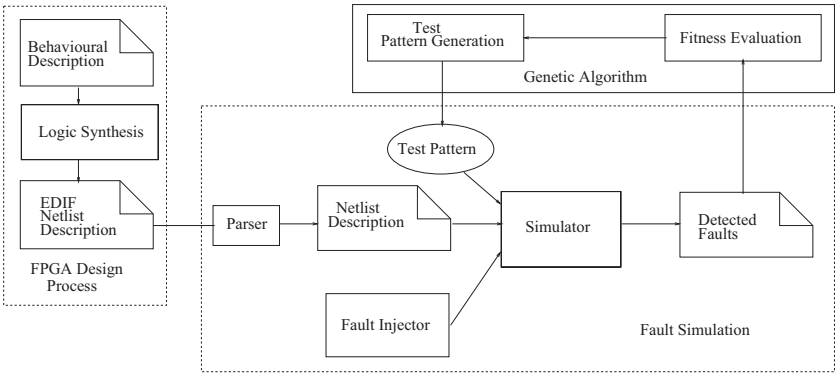


Fig. 5. The test pattern generation process.

Table 4
Circuit functions.

Circuit	Function
b01	Compare serial flows
b02	Recognize binary coded decimal numbers
b03	Resource arbiter
b06	Interrupt handler
b08	Find inclusions in sequences of numbers
b09	Serial-to-serial converter
b10	Voting system
b13	Interface to meteo sensors

In all experiments, only the excitable faults were injected. The unexcitability analysis of SEUs in the configuration memory was carried out with the SEU-X tool [64].

The GA uses an adaptive DGRT admittance threshold policy, with DGRT compaction. The GA terminates if there is no improvement in the best fitness of the population over a predetermined number of generations, or when the preset maximum number of generations is reached.

To show the behavior of the optimization process performed by the GA, we report in Fig. 6, for the ITC'99 b09 circuit, the fault coverage of the whole DGRT (i.e., the cardinality of the union of the faults detected by each DGRT entry) at each generation versus the number of generations. Here the optimization process has been tuned so as to maximize coverage, at the cost of greater test length. The figure shows how a high coverage is achieved after a small number of generations.

For the same trial considered in Fig. 6, Fig. 7 represents the number of individuals in the DGRT that detect a fault, for each generation and for each fault, before DGRT compaction. This plot shows the degree of detectability of different faults, from easily detectable (with ID's between 0 and 100 and between 250 and 347) to rarely detected or undetected ones (in the central area).

In Fig. 8, the final status of the DGRT in terms of fault coverage is shown: for each individual in DGRT and for each fault, white and black dots represents covered and uncovered faults, respectively. Black vertical lines represent undetected faults. It may be observed that many individuals are very similar in terms of detected faults.

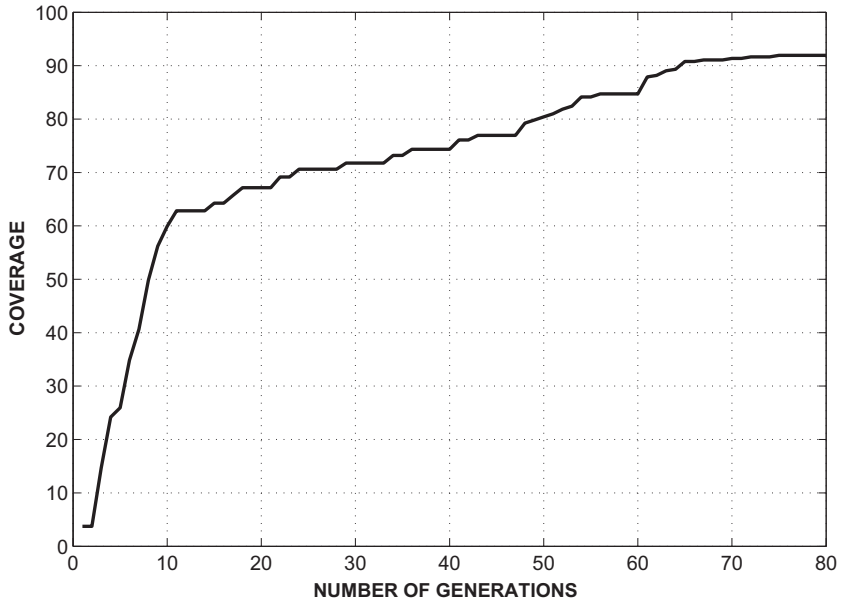


Fig. 6. DGRT coverage vs. number of generations for b09.

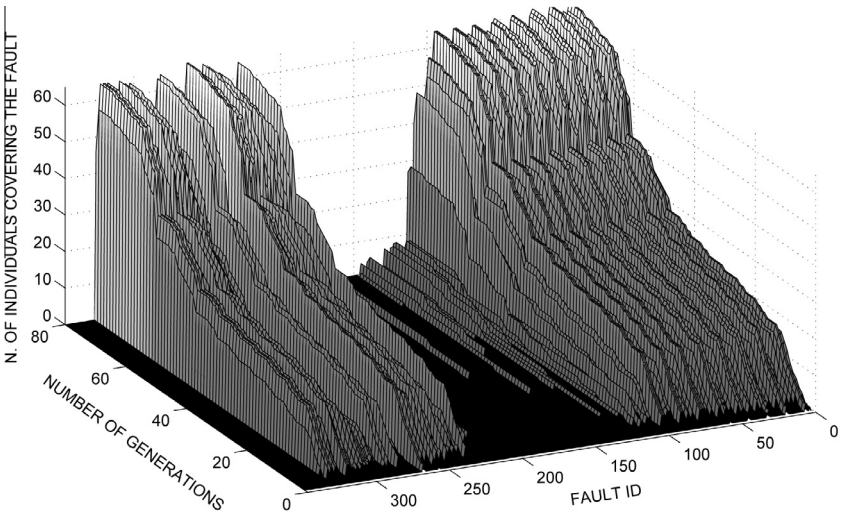


Fig. 7. Detection of faults by GA generation.

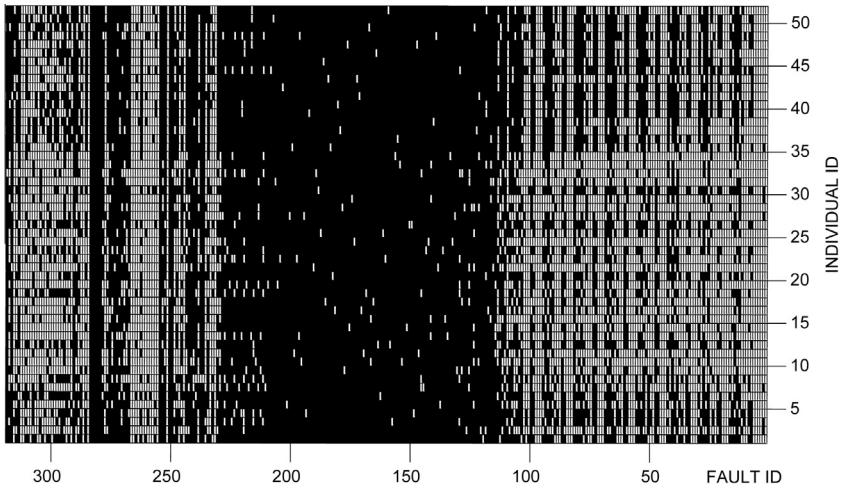


Fig. 8. Final DGRT coverage.

Table 5
Experimental results.

Circuit	Cov (%)	Length	Gen	T_{gen} (min)	Comp (%)	Time (min)
b01	100.0	94	35	0.22	43.89	7.73
b02	100.0	61	22	0.05	32.39	1.10
b03	94.8	1148	66	10.18	50.69	672.00
b06	100.0	69	13	0.41	33.10	5.38
b08	87.5	1835	94	19.19	69.15	1804.00
b09	91.9	4661	79	82.68	67.18	6532.10
b10	98.6	362	11	17.45	68.55	192.33
b13	83.3	550	18	259.83	72.38	4677.62

It has been experienced that this phenomenon occurs particularly when the fitness function is tuned to maximize coverage by loosening constraints on test length (choosing low or null values of M) and on the threshold for acceptance in the DGRT (choosing low values of $\bar{n}$). New individuals added in the DGRT may also detect faults already detected by other individuals, thus making the latter redundant. As an example, in the reported trial the compacting process reduced the length of the test set by 40.3%.

Results for the considered circuits are shown in Table 5. For each circuit, Columns *Cov*, *Length*, *Gen*, and *Time* report the measured coverage with respect to excitable faults, the test length (cumulative number of clock cycles of the test set, plus one reset cycle for each TP), the number of generations, and the simulation time, respectively. Column T_{gen} reports the average time needed to process a generation, and Column *Comp* reports the average compaction ratio, i.e., the gain in TP length achieved by the DGRT compaction algorithm.

In order to compare the genetic-based approach to other techniques, let us first consider the results presented in Table 6. Here, both random and deterministic TP generation have been performed on three different circuits. With both methods, a coverage of 100% was achieved, by progressively increasing TP length (in steps of 10000) for random generation, and by using a model-checking based method [64] for deterministic generation. It may be noted that the length of the generated TPs is consistently much greater than the one obtained with the GA, with a number of generations lower than forty. Moreover, neither the random nor the deterministic method are scalable, because they require much longer times when applied to more complex circuits than those presented in Table 6.

Table 6
Random and deterministic TP generation (100% coverage).

Circuit	Random generation		Deterministic generation	
	Length	Time (min)	Length	Time (min)
b01	100000	0.7	630	0.7
b02	10000	0.1	68	0.4
b06	100000	0.6	891	0.8

Let us now consider Table 7, which compares the results for the test patterns generated by GABES with GA (*GA* columns) with those obtained by the same tool with random testing (*Random* and *Random** columns). Times are in minutes.

Two different random testing trials were performed: In the first one (*Random* columns) a random test pattern of fixed length (10 thousand clock cycles) was used; in the second one (*Random** columns) a random test pattern with the same length as the one produced with the GA was used.

It may be observed that results obtained by the GA are much better than the ones obtained by random testing in terms of fault coverage. The time to generate the TPs is much longer with the GA, and this is due to the simulation times. The simulator performance is expected to improve with a forthcoming optimized version.

Comparing the values of *Cov* in the GA column in Table 7 with the *Random** column, the improvement, computed as the difference between *GA* and *Random** coverage, ranges between 60.28 (b03) and 87.30 (b08).

With respect to the *Random Cov* column, the improvement ranges between 0.7 (b06) and 86.9 (b08), and the solution generated by the genetic algorithm has always a shorter length.

Table 8 reports the number of stuck-at faults and the fault coverage for the considered circuits, obtained by simulation. As observed in Section 3.1, the number of faults is much smaller than in

Table 7
Comparison with random TP generation.

Circuit	GA		Random		Random∗	
	Cov (%)	Time	Cov (%)	Time	Cov (%)	Time
b01	100.0	7.73	95.70	0.30	23.40	0.020
b02	100.0	1.10	100.00	0.10	30.60	0.003
b03	94.8	672.00	50.90	33.50	33.80	5.220
b06	100.0	5.38	99.30	0.30	34.30	0.030
b08	87.5	1804.00	0.60	87.30	0.20	16.900
b09	91.9	6532.10	11.00	59.50	6.36	28.400
b10	98.6	192.33	40.00	144.42	35.05	6.160
b13	83.3	4677.62	14.22	1182.66	11.95	55.110

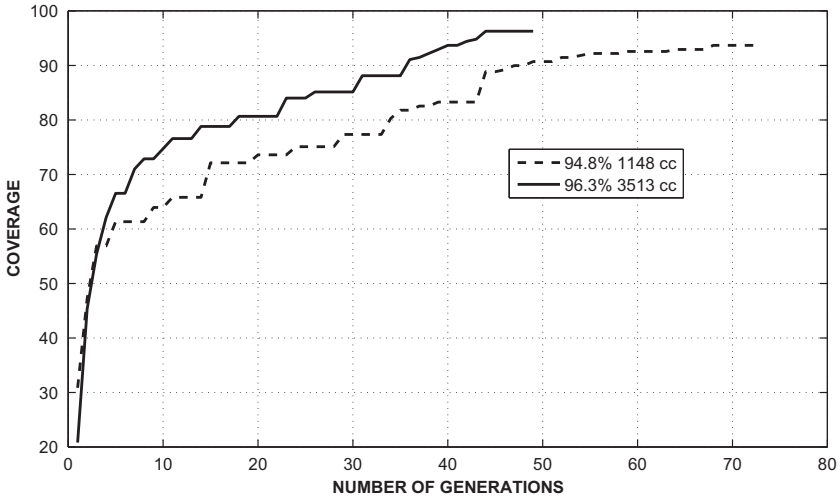


Fig. 9. Two optimization strategies for b03, in terms of unit cost per clock cycle (M).

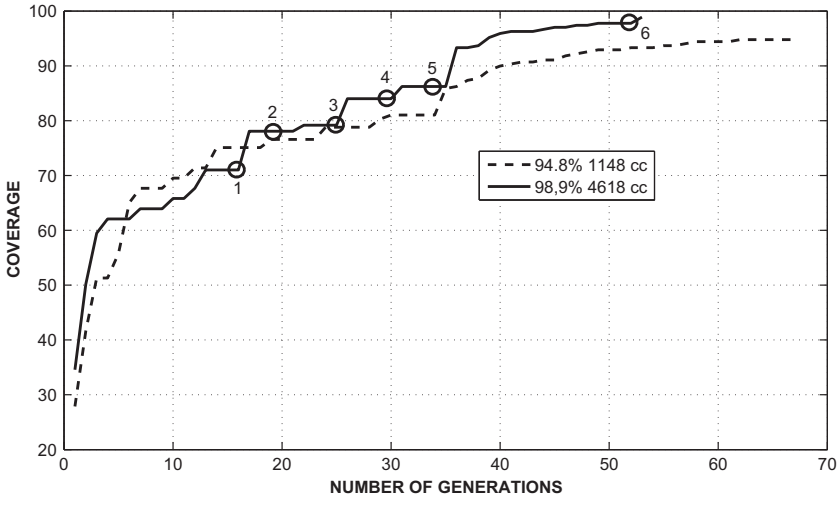


Fig. 10. Guided TP injection for b03.

the model adopted by GABES, but in spite of this, our tool achieves better coverage for this set of circuits.

Finally, Fig. 9 shows the use of GABES with b03 for different optimization strategies. Here, two trials have been carried out, with different unit cost per clock cycle (M). More specifically, the solid and dotted curves represent the coverage versus the number of generations for M = 0 and M = 1, respectively. It can be noticed that, using M = 0, the generator is able to increase the fault coverage from 94.8% to 96.3%, although producing longer test patterns, i.e., from 1148 to 3513 clock cycles.

6.1. Guided test pattern injection

In order to improve scalability of TP generation, i.e., the ability of the GA to achieve high coverage for larger circuits, the application of *partial supervision* [65] has been explored, using the circuit

Table 8
Stuck-at fault coverage.

Circuit	F_{sa}	Cov_{sa} (%)
b01	28	100.0
b02	24	100.0
b03	170	45.9
b06	36	100.0
b08	108	2.1
b09	112	37.6
b10	140	76.2
b13	254	27.1

b03 as a test case. Partial supervision is a concept from the field of machine learning, and in the context of our work it consists in injecting fresh individuals in the current generation, when the fault coverage does not improve over a small number of generations (a *local* stall). The fresh individuals are TPs that the GA is not likely to generate, and in our case they have been obtained from results of previous experiences on fault unexcitability [64]. Fig. 10 shows the simulation results for this method (solid line), compared to those obtained without TP injection (dashed line), with circles marking the points where fresh TP have been injected, after three consecutive generations without improvement in coverage. This promising technique will be object of further work.

7. Conclusions and future work

We have presented GABES, a tool for automatic test pattern generation 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 critical components of FPGA-based systems. The approach of targeting SEUs in any configuration bit of the logic resources makes our fault model very accurate. Our GA does not rely on any knowledge on the FPGA topology. Results for some circuits from the ITC'99 benchmarks have been presented. The GA shows good scalability and efficiency in terms of both fault coverage and length of test pattern.

In order to improve scalability with a genetic-based generator guided by the external behavior of the circuit, we will evaluate a guidance mechanism using pre-computed test patterns chosen by the analyst and used to stimulate the genetic exploration in unknown areas of the search space.

As further work, faults in the routing resources should be considered. We also consider enhancing the fitness function by taking into account information on error propagation in the circuit, to improve the effectiveness of the GA.

Acknowledgments

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

References

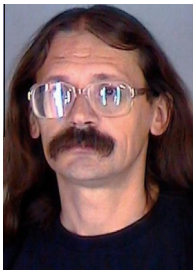
[1] R. Baumann, Radiation-induced soft errors in advanced semiconductor technologies, IEEE Transactions on Device and Materials Reliability 5 (3) (2005) 305–316.
[2] P. Graham, M. Caffrey, J. Zimmerman, D.E. Johnson, P. Sundararajan, C. Patterson, Consequences and categories of SRAM FPGA configuration SEUs, in: Proceedings of the 6th Military and Aerospace Applications of Programmable Logic Devices (MAPLD'03), 2003.
[3] J. Borecky, P. Kubalik, H. Kubatova, Reliable railway station system based on regular structure implemented in FPGA, in: Proceedings of the 12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools (DSD '09), 2009, pp. 348–354.
[4] V. Winkler, J. Detlefsen, U. Siart, J. Buchler, M. Wagner, FPGA-based signal processing of an automotive radar sensor, in: Proceedings of the First European Radar Conference (EURAD), 2004, pp. 245–248.
[5] J. Henaut, D. Dragomirescu, R. Plana, FPGA based high date rate radio interfaces for aerospace wireless sensor systems, in: Proceedings of the Fourth International Conference on Systems (ICONS '09), 2009, pp. 173–178.
[6] I.O. for Standardization (ISO), 26262-5: Road vehicles - Functional safety - Part 5. Product development: hardware level, draft (December 2009).
[7] E.C. for Electrotechnical Standardization (CENELEC), EN 50129: Railway Applications - Communications, Signaling and Processing Systems - Safety Related Electronic Systems for Signaling, February 2003.
[8] I.A.E.A. (IAEA), NS-G-1.3: Instrumentation and Control Systems Important to Safety in Nuclear Power Plants, IAEA Safety Standards Series, 2002.
[9] W. Huang, F. Meyer, N. Park, F. Lombardi, Testing memory modules in SRAM-based configurable FPGAs, in: Proceedings of the International Workshop on Memory Technology, Design and Testing, 1997, pp. 79–86.
[10] M. Renovell, J. Portal, J. Figuras, Y. Zorian, Minimizing the number of test configurations for different FPGA families, in: Proceedings of the Eighth Asian Test Symposium (ATS '99), 1999, pp. 363–368.
[11] C. Stroud, S. Wijesuriya, C. Hamilton, M. Abramovici, Built-in self-test of FPGA interconnect, in: Proceedings of the International Test Conference, 1998, pp. 404–411.
[12] M. Rozkovec, J. Jenicek, O. Novak, Application dependent FPGA testing method, in: Proceedings of the 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools (DSD10), 2010, pp. 525–530.
[13] M. Tahoori, Application-dependent testing of FPGAs, IEEE Transactions on Very Large Scale Integration (VLSI) Systems 14 (9) (2006) 1024–1033.
[14] F. Corno, P. Prinetto, M. Rebaudengo, 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) (1996) 991–1000.
[15] T. Garbolino, G. Papa, Genetic algorithm for test pattern generator design, Applied Intelligence 32 (2) (2010) 193–204.
[16] [16] S. Nalini, S. Sivakumar, Test pattern generation for relaxed n-detect test sets using genetic algorithm, International Journal of Advanced, Information Science and Technology 12 (12) (2013).
[17] M. Sangeeta, M. Chopra, Mr.H.P.S.Dhami, Generating boolean SAT based test pattern generation using multi-objective genetic algorithm, International Journal of Computer Applications 6 (8) (2010) 1–4, published By Foundation of Computer Science.
[18] M. Abramovici, M.A. Breuer, A.D. Friedman, Digital Systems Testing and Testable Design, John Wiley & Sons., 1990.
[19] M. Renovell, J. Portal, P. Faure, J. Figueras, Y. Zorian, Analyzing the test generation problem for an application-oriented test of FPGAs, in: Proceedings of the IEEE European Test, Workshop, 2000, pp. 75–80.
[20] M. Rebaudengo, M. Sonza Reorda, 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), 2002, pp. 372–380.
[21] G. Papa, T. Garbolino, F. Novak, A. Hławiczka, Deterministic test pattern generator design with genetic algorithm approach, Journal of Electrical Engineering 58 (3) (2007) 121–127.
[22] A. Fin, F. Fummi, Genetic algorithms: the philosopher's stone or an effective solution for high-level TPG?, in: High-Level Design Validation and Test Workshop, 2003. Eighth IEEE International, 2003, pp. 163–168.
[23] C. Bernardeschi, L. Cassano, A. Domenici, G. Gennaro, 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.
[24] C. Bernardeschi, L. Cassano, A. Domenici, Failure probability and fault observability of SRAM-FPGA systems, in: International Conference on Field Programmable Logic and Applications (FPL 2011), 2011, pp. 385–388.
[25] C. Bernardeschi, L. Cassano, A. Domenici, Failure probability of SRAM-FPGA systems with stochastic activity networks, in: Proceedings of the 14th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems, 2011.
[26] M. Ceschia, M. Bellato, A. Paccagnella, S. C. Lee, C. Wan, A. Kaminski, M. Menichelli, A. Papi, J. Wyss, Ion beam testing of Altera Apex FPGAs, in: Proceedings of the 2002 IEEE Radiation Effects Data Workshop, 2002, pp. 45–50.
[27] M. Straka, J. Kastil, Z. Kotasek, SEU Simulation framework for Xilinx FPGA: first step towards testing fault tolerant systems, in: Proceedings of the 14th Euromicro Conference on Digital System Design (DSD2011), 2011, pp. 223–230.
[28] G. Asadi, M.B. Tahoori, An analytical approach for soft error rate estimation of SRAM-based FPGAs, in: Proceedings of the Military and Aerospace Applications of Programmable Logic Devices (MAPLD), 2004, pp. 2991–2994.
[29] D.G. Gutiérrez, Single event upsets simulation tool functional description, Tech. Rep. TEC-EDM/DGG-SST2, ESA-ESTEC, 2000.
[30] F.L. Kastensmidt, L. Carro, R. Reis, Fault-Tolerance Techniques for SRAM-Based FPGAs (Frontiers in Electronic Testing), Springer–Verlag New York Inc., Secaucus, NJ, USA, 2006.
[31] L. Rockett, D. Patel, S. Danziger, B. Cronquist, J. WANG, Radiation hardened fpga technology for space applications, in: Aerospace Conference, 2007 IEEE, 2007, pp. 1–7.

[32] Using EDAC RAM for RadTolerant RTAX-S/SL and Axcelerator FPGAs, application Note AC319, 2008.
[33] J.P. Roth, Diagnosis of automata failures: a calculus and a method, IBM Journal of Research and Development 10 (4) (1966) 278–291.
[34] P. Goel, An implicit enumeration algorithm to generate tests for combinational logic circuits, IEEE Transactions on Computers 30 (3) (1981) 215–222.
[35] H. Fujiwara, T. Shimono, On the acceleration of test generation algorithms, IEEE Transactions on Computers 32 (12) (1983) 1137–1144.
[36] K.D. Wagner, C.K. Chin, E.J. McCluskey, Pseudorandom testing, IEEE Transactions on Computers 36 (3) (1987) 332–343.
[37] R. Jiann-Chyi, H. Ying-Fu, W. Po-Han, A novel reseeding mechanism for pseudo-random testing of VLSI circuits, in: IEEE International Symposium on Circuits and Systems (ISCAS 2005), vol. 3, 2005, pp. 2979–2982.
[38] A. Al-Yamani, S. Mitra, E. McCluskey, Optimized reseeding by seed ordering and encoding, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 24 (2) (2005) 264–270.
[39] P. Goel, RAPS Test Pattern Generator, IBM Technical Disclosure Bulletin 21 (7) (1978) 2787–2791.
[40] M. Abramovici, J.J. Kulikowski, P.R. Menon, D.T. Miller, Smart and fast: Test generation for VLSI scan-design circuits, IEEE Design and Test 3 (4) (1986) 43–54.
[41] C. Ioannides, K.I. Eder, Coverage-directed test generation automated by machine learning — a review, ACM Transactions on Design Automation of Electronic Systems 17 (1) (2012) 7:1–7:21.
[42] A. Samarah, A. Habibi, S. Tahar, N. Kharma, Automated coverage directed test generation using a cell-based genetic algorithm, in: High-Level Design Validation and Test Workshop, 2006, Eleventh Annual IEEE International, 2006, pp. 19–26.
[43] Z. Michalewicz, Genetic Algorithms Plus Data Structures Equals Evolution Programs, 2nd Edition., Springer–Verlag New York Inc., Secaucus, NJ, USA, 1994.
[44] M. Gen, R. Cheng, Genetic Algorithms and Engineering Design, John Wiley & Sons., 1997.
[45] F. Ferrandi, A. Fin, F. Fummi, D. Sciuto, Functional test generation: overview and proposal of a hybrid genetic approach, in: R. Drechsler, N. Drechsler (Eds.), Evolutionary Algorithms in Circuit Design, Kluwer, 2003, pp. 105–142.
[46] D.G. Saab, Y.G. Saab, J.A. Abraham, Cris: a test cultivation program for sequential VLSI circuits, in: 1992 IEEE/ACM International Conference Proceedings on Computer-Aided Design, ICCAD '92, IEEE Computer Society Press, Los Alamitos, CA, USA, 1992, pp. 216–219.
[47] E. Rudnick, J.G. Holm, D.G. Saab, J.H. Patel, Application of simple genetic algorithms to sequential circuit test generation, in: Proc. European Design and Test Conf, 1994, pp. 40–45.
[48] E. Rudnick, J. Patel, G. Greenstein, T. Niermann, A genetic algorithm framework for test generation, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 16 (9) (1997) 1034–1044.
[49] Y. Skobtsov, V. 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), 2010, pp. 286–291.
[50] F. Corno, G. Cumani, M. Sonza Reorda, G. Squillero, Fully automatic test program generation for microprocessor cores, in: Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1, DATE '03, IEEE Computer Society, Washington, DC, USA, 2003, pp. 1006–1011.
[51] F. Corno, E. Sánchez, M.S. Reorda, G. Squillero, Automatic test program generation: a case study, IEEE Design and Test 21 (2) (2004) 102–109.
[52] F. Corno, E. Sanchez, M.S. Reorda, G. Squillero, Code generation for functional validation of pipelined microprocessors, Journal of Electronic Testing 20 (3) (2004) 269–278.
[53] I. Kuon, R. Tessier, J. Rose, Fpga Architecture: Survey and Challenges, 2008.
[54] M. O'Dare, T. Arslan, Hierarchical test pattern generation using a genetic algorithm with a dynamic global reference table, in: Genetic Algorithms in Engineering Systems: Innovations and Applications, 1995. GALESIA. First International Conference on (Conf. Publ. No. 414), 1995, pp. 517–523.
[55] J.H. Holland, Adaptation in Natural and Artificial Systems, The University of Michigan Press, Ann Arbor, 1975.
[56] F. Corno, M. Sonza Reorda, G. Squillero, RT-Level ITC'99 Benchmarks and First ATPG Results, IEEE Design and Test 17 (2000) 44–53.
[57] D. Wolpert, W. Macready, No free lunch theorems for optimization, Evolutionary Computation, IEEE Transactions on 1 (1) (1997) 67–82.
[58] F. Pinel, G. Danoy, P. Bouvry, Evolutionary algorithm parameter tuning with sensitivity analysis, in: Proceedings of the 2011 International Conference on Security and Intelligent Information Systems, SIIS'11, Springer–Verlag, Berlin, Heidelberg, 2012, pp. 204–216.
[59] R. Kumar, Jyotishree, Blending roulette wheel selection & rank selection in genetic algorithms, International Journal of Machine Learning and Computing 2 (4) (2012) 365–370.
[60] W.H. Sanders, J.F. Meyer, Stochastic activity networks: formal definitions and concepts, in: Lectures on Formal Methods and Performance Analysis: First EEF/Euro Summer School on Trends in Computer Science, Springer–Verlag New York Inc., New York, NY, USA, 2002, pp. 315–343.
[61] D.D. Deavours, G. Clark, T. Courtney, D. Daly, S. Derisavi, J.M. Doyle, W.H. Sanders, P.G. Webster, The Möbius framework and its implementation, IEEE Transactions on Software Engineering 28 (10) (2002) 956–969.
[62] Electronic design interchange format (EDIF) - Part 2: Version 4 0 0, Tech. Rep. IEC 61690-2, IEC, International Electrotechnical Commission, 2000.
[63] ISE design suite software manuals and help, 2010. Available from: http://www.xilinx.com/support/documentation/swmanuals
[64] C. Bernardeschi, L. Cassano, A. Domenici, SEU-X: a SEu Un-eXcitability prover for SRAM-FPGAs, in: 18-th IEEE International On-Line Testing Symposium (IOLTS), 2012, pp. 29–34.
[65] W. Pedrycz, J. Waletzky, Fuzzy clustering with partial supervision, systems, man, and cybernetics, part B: cybernetics, IEEE Transactions on 27 (5) (1997) 787–795.

Cinzia Bernardeschi received the Laurea degree in Computer Science in 1987 and the PhD degree in 1996, both from the University of Pisa.
She is Associate Professor with the Department of Information Engineering of the University of Pisa. Her research interests are in the area of software engineering, dependable systems and application of formal methods for specification and verification of safety-critical systems.

Luca Cassano received the Bachelor's degree and the Master's degree in Computer Engineering in 2006 and 2009 respectively from the University of Pisa, Italy. In 2013 he received the Ph.D. in Information Engineering from the Department of Information Engineering of the University of Pisa. During the Ph.D. course he spent a visiting period at the Department of Automation and Informatics of the Politecnico di Torino, Italy and a visiting period at the Cognitive Interaction Technology - Center of Excellence (CITEC) of the University of Bielefeld, Germany. He is currently a post doctoral research fellow at the Department of Information Engineering of the University of Pisa and at the Department of Electronics, Informatics and Bioengineering of the Politecnico di Milano. His research interests are mainly focused on the use of formal and semi formal methods for fault simulation, test pattern generation and diagnosis of electronic circuits and systems.

Mario G.C.A. Cimino received the Ph.D. degree in Information Engineering from the University of Pisa (Italy) in 2007. In 2006, he spent six months as a visiting PhD student in the Electrical & Computer Engineering Research Facility of the University of Alberta, Edmonton (Canada). He co-organized three editions of the Workshop on Computational Intelligence for Personalization in Web Content and Service Delivery. He is reviewer of research projects funded by the Czech Science Foundation. From 2008 to 2012 he was involed, as a Postdoctoral Research Fellow, in the fields of Mobile Information Systems, Business Process Analysis and Computational Intelligence. Since June 2012, he is with the Department of Information Engineering of the University of Pisa as a Researcher in Information Systems. He is (co-) author of about 30 publications.



Andrea Domenici obtained his PhD in Information Engineering in 1992 with a thesis on the implementation of the Gödel logic programming language on parallel machines. He has been an assistant professor at the Sant'Anna School of University Studies and Doctoral Research, Pisa, and he is currently at the Department of Information Engineering of the University of Pisa, where he teaches Software Engineering and does research in the fields of dependable systems and application of formal methods in the development of safety- and mission-critical systems. He has also been active in Object-oriented design and Grid architectures. In this latter field, he took part in the European Datagrid Project and in the EGEE project, in cooperation with CERN and the Italian National Institute of Nuclear Physics (INFN).