GABES: a Genetic Algorithm Based Environment for SEU Testing in SRAM-FPGAs

C. Bernardeschi, L. Cassano, M. G.C.A. Cimino, A. Domenici
Department of Information Engineering
University of Pisa, Italy

Abstract

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.

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], CEN-ELEC 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, 10, 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, 15, 16, 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 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: Sect. 2 is a brief survey of works addressing test pattern generation in digital circuits; in Sect. 3 the considered fault model is presented, fault simulation is briefly discussed and basic concepts of evolutionary approaches to problem solving are introduced; in Sect. 4 the main characteristics of the proposed GA are discussed; Sect. 5 briefly presents the structure of the test pattern generation tool; Sect. 6 presents the results of the proposed algorithm for some circuits from the ITC’99 benchmarks; Sect. 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, 44, 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. Figure 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 $y_f = x_1 \cdot x_2 + x_3 \cdot x_4 + x_1 \cdot x_2 \cdot x_3 \cdot x_4$. 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).

Figure 2 shows LUT L from Figure 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 \textit{propagate} to the primary outputs. Again, error propagation depends on the structure and state of C.

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

### 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 Figure 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 Figure 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 \( \lambda \).

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, \( p_m \), 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 \( p_m \) are in the order of \( 10^{-2} \) [43]. We control the mutation operator by a dynamic \( p_m \) which decreases linearly between an initial value \( p_m \) and a value \( p_m \) at the final generation.

With a linearly decreasing \( p_m \), 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 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:

\[
c_i = \sum_{j=1}^{n_i} \left( \xi_{ij} + 1 \right)^{-k},
\]

where \(n_i\) is the number of faults detected by the \(i\)-th test pattern; \(\xi_{ij}\) is the number of test patterns, generated before the \(i\)-th one, that detect fault \(j\); \(k\) 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 (Sec. 4.5). The table also reports the values assigned to the parameters in the experiments discussed in Sect. 6.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\lambda)</td>
<td>rate parameter for cut-point distribution</td>
<td>1</td>
</tr>
<tr>
<td>(p_c)</td>
<td>crossover rate</td>
<td>0.8</td>
</tr>
<tr>
<td>(p_M)</td>
<td>maximum mutation rate</td>
<td>0.15</td>
</tr>
<tr>
<td>(p_m)</td>
<td>minimum mutation rate</td>
<td>0.05</td>
</tr>
<tr>
<td>(N)</td>
<td>number of possible faults</td>
<td>–</td>
</tr>
<tr>
<td>(M)</td>
<td>penalty coefficient for TP length</td>
<td>0.5</td>
</tr>
<tr>
<td>(L)</td>
<td>maximum length for TPs</td>
<td>10000</td>
</tr>
<tr>
<td>(k)</td>
<td>penalty exponent for easy-to-detect faults</td>
<td>0.75</td>
</tr>
<tr>
<td>(s_{\text{max}})</td>
<td>maximum number of iterations</td>
<td>2000</td>
</tr>
<tr>
<td>(S)</td>
<td>TP population size</td>
<td>200</td>
</tr>
<tr>
<td>(m)</td>
<td>number of iterations considered for stall condition</td>
<td>20</td>
</tr>
<tr>
<td>(\bar{n})</td>
<td>threshold for acceptance into DGRT</td>
<td>20</td>
</tr>
<tr>
<td>(m_d)</td>
<td>number of iterations considered for (\bar{n}) adaptation</td>
<td>20</td>
</tr>
</tbody>
</table>

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 $\xi_{ij}$ of other individuals that have been shown to detect the fault before $i$. The number $\xi_{ij}$ 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 $s_{\text{max}}$ 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 $\text{improve}(m)$ 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.

```plaintext
s ← 0; D ← ∅; D' ← ∅
P_0 ← (S randomly generated test patterns)
while coverage(D) < N ∧ \text{improve}(m) ∧ s < s_{\text{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($i, \bar{n}, D$) then
      D ← D ∪ \text{new}($i, \bar{n}, D$)
    end if
  end for
  if $s < s_{\text{max}}$ then
    s ← s + 1; P_s ← \text{ga}(P_{s-1}, D)
  end if
end while
D' ← \text{compact}(D)
return \text{testset}(D')
```

Algorithm 1: The overall algorithm.

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 \(\text{finds}(i, \bar{n}, D)\) 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 \(\text{new}(i, \bar{n}, D)\) returns the pairs \(\langle i, j_1 \rangle, \ldots, \langle i, j_{\bar{n}} \rangle\) such that test pattern \(i\) detects a fault in that set. The value of \(\bar{n}\) is set initially at 20 circuits. 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 \(\text{ga}(P_{s-1}, D)\) produces the new generation \(P_s\) from the previous one, using the DGRT to compute the fitness. On exit from the outermost loop, function \(\text{compact}(D)\) produces a new DGRT by removing individuals, or terminal segments thereof, whose faults are covered by others. Finally, \(\text{testset}(D)\) 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 \(s_{\text{max}}\) is reached.

\[
\begin{align*}
P_M & \leftarrow \emptyset; A \leftarrow \emptyset; B \leftarrow \emptyset \\
& \text{for } i = 1 \text{ to } Q \text{ do} \\
& \quad x \leftarrow \text{select}(P); P_M \leftarrow P_M \cup \{x\} \\
& \text{end for} \\
& \text{for } i = 1 \text{ to } Q/2 \text{ do} \\
& \quad (x, y) \leftarrow \text{pair}(P_M); (x', y') \leftarrow \text{crossover}(x, y, \lambda) \\
& \quad x'' \leftarrow \text{mutate}(x', p_\mu); y'' \leftarrow \text{mutate}(y', p_\mu) \\
& \quad A \leftarrow A \cup \{x'', y''\} \\
& \text{end for} \\
& \text{for } i = 1 \text{ to } S - Q \text{ do} \\
& \quad x \leftarrow \text{select}(P, p_c); \\
& \quad B \leftarrow B \cup \{x\} \\
& \text{end for} \\
& P \leftarrow A \cup B \\
& \text{return } P
\end{align*}
\]

Algorithm 2: The genetic algorithm.

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 (Sec. 3.3).

Function \(\text{select}(P)\) returns an individual from \(P\), selected with the roulette wheel method, and function \(\text{pair}(P_M)\) returns a pair of parents from \(P_M\), selected with the roulette wheel method. Function \(\text{crossover}(x, y, \lambda)\) returns the offspring of a pair of parents, with \(\lambda\) as the level of exploitation parameter for cut point selection (Sec. 4.2). Mutation is then applied to the selected parents with probability \(p_\mu\), 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 (Sect. 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 $M$, $L$, $s_{\text{max}}$, $S$) 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 $\lambda$, 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, $p_{\mu}$ and $p_{\mu}'$, 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 (Sec. 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.
Table 2: Genetic parameters and related performance.

<table>
<thead>
<tr>
<th>$\bar{n}$</th>
<th>$m_d$</th>
<th>gen</th>
<th>len</th>
<th>time</th>
<th>gen</th>
<th>len</th>
<th>time</th>
<th>gen</th>
<th>len</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td>17</td>
<td>131</td>
<td>3.17</td>
<td>6</td>
<td>57</td>
<td><strong>0.22</strong></td>
<td>9</td>
<td>101</td>
<td>2.98</td>
</tr>
<tr>
<td>5</td>
<td>25</td>
<td>14</td>
<td>145</td>
<td><strong>2.27</strong></td>
<td>6</td>
<td>77</td>
<td>0.25</td>
<td>10</td>
<td>112</td>
<td>3.63</td>
</tr>
<tr>
<td>5</td>
<td>35</td>
<td>18</td>
<td>126</td>
<td>2.60</td>
<td>6</td>
<td>73</td>
<td>0.23</td>
<td>6</td>
<td>92</td>
<td>2.12</td>
</tr>
<tr>
<td>5</td>
<td>50</td>
<td>18</td>
<td>140</td>
<td>3.27</td>
<td>14</td>
<td>76</td>
<td>0.62</td>
<td>5</td>
<td>96</td>
<td><strong>1.73</strong></td>
</tr>
<tr>
<td>10</td>
<td>5</td>
<td>14</td>
<td>119</td>
<td>2.52</td>
<td>11</td>
<td>106</td>
<td>0.48</td>
<td>9</td>
<td>96</td>
<td>3.43</td>
</tr>
<tr>
<td>10</td>
<td>25</td>
<td>27</td>
<td>101</td>
<td>5.18</td>
<td>29</td>
<td>52</td>
<td>1.53</td>
<td>28</td>
<td>99</td>
<td>13.13</td>
</tr>
<tr>
<td>10</td>
<td>35</td>
<td>46</td>
<td>141</td>
<td>10.07</td>
<td>12</td>
<td>76</td>
<td>0.55</td>
<td>9</td>
<td>88</td>
<td><strong>3.38</strong></td>
</tr>
<tr>
<td>10</td>
<td>50</td>
<td>45</td>
<td>127</td>
<td>10.25</td>
<td>19</td>
<td>67</td>
<td>0.95</td>
<td>18</td>
<td>122</td>
<td>7.38</td>
</tr>
<tr>
<td>15</td>
<td>5</td>
<td>16</td>
<td>112</td>
<td><strong>3.10</strong></td>
<td>14</td>
<td>57</td>
<td>0.60</td>
<td>8</td>
<td>72</td>
<td><strong>2.83</strong></td>
</tr>
<tr>
<td>15</td>
<td>25</td>
<td>41</td>
<td><strong>103</strong></td>
<td>9.33</td>
<td>21</td>
<td>52</td>
<td>1.08</td>
<td>14</td>
<td>102</td>
<td>6.12</td>
</tr>
<tr>
<td>15</td>
<td>35</td>
<td>43</td>
<td>141</td>
<td>10.22</td>
<td>18</td>
<td>61</td>
<td>0.80</td>
<td>38</td>
<td>87</td>
<td>20.00</td>
</tr>
<tr>
<td>15</td>
<td>50</td>
<td>64</td>
<td>120</td>
<td>17.38</td>
<td>23</td>
<td>95</td>
<td>1.10</td>
<td>13</td>
<td>108</td>
<td>5.35</td>
</tr>
<tr>
<td>20</td>
<td>5</td>
<td>20</td>
<td>147</td>
<td>4.07</td>
<td>11</td>
<td>54</td>
<td>0.47</td>
<td>11</td>
<td>69</td>
<td>4.23</td>
</tr>
<tr>
<td>20</td>
<td>25</td>
<td>48</td>
<td><strong>117</strong></td>
<td>11.13</td>
<td>14</td>
<td>81</td>
<td>0.63</td>
<td>28</td>
<td>97</td>
<td>13.37</td>
</tr>
<tr>
<td>20</td>
<td>35</td>
<td>44</td>
<td>143</td>
<td>10.30</td>
<td>12</td>
<td>78</td>
<td>0.53</td>
<td>17</td>
<td>113</td>
<td>7.37</td>
</tr>
<tr>
<td>20</td>
<td>50</td>
<td>48</td>
<td>119</td>
<td>13.20</td>
<td>7</td>
<td>49</td>
<td>0.30</td>
<td>42</td>
<td><strong>64</strong></td>
<td>21.50</td>
</tr>
<tr>
<td>30</td>
<td>5</td>
<td>24</td>
<td>164</td>
<td><strong>5.10</strong></td>
<td>15</td>
<td>60</td>
<td>0.62</td>
<td>12</td>
<td>69</td>
<td><strong>4.38</strong></td>
</tr>
<tr>
<td>30</td>
<td>25</td>
<td>27</td>
<td>109</td>
<td>6.25</td>
<td>29</td>
<td>70</td>
<td>1.53</td>
<td>29</td>
<td>86</td>
<td>13.95</td>
</tr>
<tr>
<td>30</td>
<td>35</td>
<td>52</td>
<td>107</td>
<td>14.00</td>
<td>27</td>
<td><strong>39</strong></td>
<td>1.45</td>
<td>23</td>
<td>96</td>
<td>10.47</td>
</tr>
<tr>
<td>30</td>
<td>50</td>
<td>23</td>
<td><strong>105</strong></td>
<td>5.12</td>
<td>19</td>
<td>45</td>
<td>0.88</td>
<td>48</td>
<td>85</td>
<td>25.22</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>19</td>
<td>101</td>
<td><strong>3.27</strong></td>
<td>16</td>
<td>70</td>
<td>0.78</td>
<td>12</td>
<td>96</td>
<td><strong>4.48</strong></td>
</tr>
<tr>
<td>15</td>
<td>20</td>
<td>30</td>
<td>112</td>
<td>5.47</td>
<td>19</td>
<td>73</td>
<td>0.87</td>
<td>13</td>
<td>69</td>
<td><strong>5.38</strong></td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>35</td>
<td><strong>94</strong></td>
<td>7.73</td>
<td>22</td>
<td><strong>61</strong></td>
<td>1.10</td>
<td>22</td>
<td>124</td>
<td>10.42</td>
</tr>
<tr>
<td>30</td>
<td>20</td>
<td>32</td>
<td>177</td>
<td>7.57</td>
<td>24</td>
<td>63</td>
<td>1.27</td>
<td>27</td>
<td>94</td>
<td>12.70</td>
</tr>
</tbody>
</table>

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.

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 M"obius [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.

![Figure 5: The Test Pattern Generation process.](image-url)

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

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

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

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>b01</td>
<td>Compare serial flows</td>
</tr>
<tr>
<td>b02</td>
<td>Recognize binary coded decimal numbers</td>
</tr>
<tr>
<td>b03</td>
<td>Resource arbiter</td>
</tr>
<tr>
<td>b06</td>
<td>Interrupt handler</td>
</tr>
<tr>
<td>b08</td>
<td>Find inclusions in sequences of numbers</td>
</tr>
<tr>
<td>b09</td>
<td>Serial-to-serial converter</td>
</tr>
<tr>
<td>b10</td>
<td>Voting system</td>
</tr>
<tr>
<td>b13</td>
<td>Interface to meteo sensors</td>
</tr>
</tbody>
</table>
(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 Sec. 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.

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.

![Figure 6: DGRT coverage vs. number of generations for b09.](image)

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 represent 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. 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
Figure 8: Final DGRT coverage.

Table 5: Experimental results.

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

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 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).

<table>
<thead>
<tr>
<th>Circuit</th>
<th>Random generation</th>
<th>Deterministic generation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Length</td>
<td>Time (min)</td>
</tr>
<tr>
<td>b01</td>
<td>100000</td>
<td>0.7</td>
</tr>
<tr>
<td>b02</td>
<td>10000</td>
<td>0.1</td>
</tr>
<tr>
<td>b06</td>
<td>100000</td>
<td>0.6</td>
</tr>
</tbody>
</table>

Table 7: Comparison with random TP generation.

<table>
<thead>
<tr>
<th>Circuit</th>
<th>GA</th>
<th>Random</th>
<th>Random*</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Cov (%)</td>
<td>Time</td>
<td>Cov (%)</td>
</tr>
<tr>
<td>b01</td>
<td>100.0</td>
<td>7.73</td>
<td>95.70</td>
</tr>
<tr>
<td>b02</td>
<td>100.0</td>
<td>1.10</td>
<td>100.00</td>
</tr>
<tr>
<td>b03</td>
<td>94.8</td>
<td>672.00</td>
<td>50.90</td>
</tr>
<tr>
<td>b06</td>
<td>100.0</td>
<td>5.38</td>
<td>99.30</td>
</tr>
<tr>
<td>b08</td>
<td>87.5</td>
<td>1804.00</td>
<td>0.60</td>
</tr>
<tr>
<td>b09</td>
<td>91.9</td>
<td>6532.10</td>
<td>11.00</td>
</tr>
<tr>
<td>b10</td>
<td>98.6</td>
<td>192.33</td>
<td>40.00</td>
</tr>
<tr>
<td>b13</td>
<td>83.3</td>
<td>4677.62</td>
<td>14.22</td>
</tr>
</tbody>
</table>

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 Sec. 3.1, the number of faults is much smaller than in the model adopted by GABES, but in spite of this, our tool achieves better coverage for this set of circuits.

Finally, Figure 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.
Table 8: Stuck-at Fault Coverage.

<table>
<thead>
<tr>
<th>Circuit</th>
<th>$F_{sa}$</th>
<th>Cov$_{sa}$ (%)</th>
<th>Circuit</th>
<th>$F_{sa}$</th>
<th>Cov$_{sa}$ (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>b01</td>
<td>28</td>
<td>100.0</td>
<td>b08</td>
<td>108</td>
<td>2.1</td>
</tr>
<tr>
<td>b02</td>
<td>24</td>
<td>100.0</td>
<td>b09</td>
<td>112</td>
<td>37.6</td>
</tr>
<tr>
<td>b03</td>
<td>170</td>
<td>45.9</td>
<td>b10</td>
<td>140</td>
<td>76.2</td>
</tr>
</tbody>
</table>

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

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 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]. Figure 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.

References


