Experiences in the Use of Evolutionary Techniques for Testing Digital Circuits

F. Corno, M. Rebaudengo, M. Sonza Reorda

Politecnico di Torino
Dipartimento di Automatica e Informatica
Torino, Italy
http://www.cad.polito.it/

ABSTRACT

The generation of test patterns for sequential circuits is one of the most challenging problems arising in the field of Computer-Aided Design for VLSI circuits. In the past decade, Genetic Algorithms have been deeply investigated as a possible approach: several algorithms have been described, and significant improvements have been proposed with respect to their original versions. As a result, Genetic Algorithm-based test pattern generators can now effectively compete with other methods, such as topological or symbolic ones. This paper discusses the advantages and disadvantages of GA-based approaches and describes GATTO, a state-of-the-art Genetic Algorithm-based test pattern generator. Other algorithms belonging to the same category are outlined as well. The paper puts GATTO and other GA-based tools in perspective, and shows that Evolutionary Computation techniques can successfully compete with more traditional approaches, or be integrated with them.

Keywords: Test Pattern Generation, Genetic Algorithms, Computer-Aided Design, VLSI circuits

1 INTRODUCTION

Testing a VLSI circuit is often performed through the application to its Input Pins of a sequence of values, such that the values that one can observe on the Output Pins of the fault-free circuit are different from the ones appearing on the same Pins of any faulty circuit. One of the main problems is how to generate a suitable sequence of values to be applied to the Input Pins (Automatic Test Pattern Generation, or *ATPG*).

The economical importance of ATPG tools for digital circuits is continuously growing, and the demand for efficient algorithms and tools able to handle the current circuits is thus very strong. Correspondingly, there have been significant research efforts in this field, which produced tens of proposals in terms of ATPG algorithms and techniques. In the last years, one of the main goals of these efforts was to develop effective algorithms for sequential circuits.

Due to the great increase in the circuit size and complexity, this task is now critical from the point of view of the required computational power, and a significant amount of research activities has been devoted to it in the past years[1]. The *stuck-at* fault model has generally been adopted, and efficient algorithms have been proposed for combinational networks. On the opposite side, the problem of effectively dealing with very large sequential circuits is still open, even if some commercial tools are already available. In fact, no proposed method is able to successfully handle the complete set of real-world circuits test engineers have to face with: they either definitely get off or generate very poor results.

Generating test sequences for synchronous sequential circuits (Fig. 1) is a challenging problem for several reasons, which concur to make it harder than the corresponding problem for combinational circuits:

- each fault must be first excited (i.e., a value different from the one forced by the fault must be propagated to the fault source), and this goal is generally reached when a given value is present not only on the Primary Inputs (PIs), but also on the Flip-Flop outputs (Pseudo-Primary Inputs or PPIs); a certain number of clock cycles can thus be devoted to force the Flip-Flop outputs to the required value (*Excitation Sequence*);
- the difference existing on the fault source between the values of the faulty-free and the faulty circuit must be then propagated to the Primary Outputs (POs): this goal can seldom be reached in the same time frame which excites the

fault; more often, the fault is first propagated to the Flip-Flop inputs (Pseudo-Primary Outputs or PPOs), and a given number of clock cycles is required to propagate the fault effects from the Flip-Flops to the Primary Outputs (*Propagation Sequence*);

- neither the length of both the Excitation and the Propagation Sequence is known a priori, nor any meaningful upper bound can be found for this length; this fact makes really enormous the size of the search space to be analyzed by the algorithms;
- untestable faults, i.e., faults for which no test sequence exists, require a big amount of computation to be recognized.

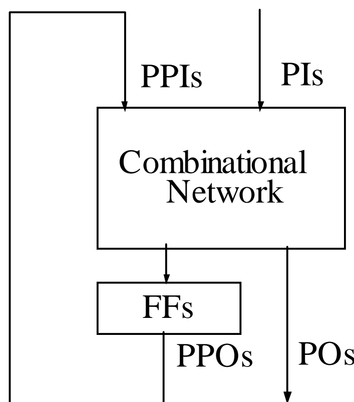

Figure 1: A synchronous sequential circuit.

Different approaches have been proposed to solve the problem of Automatic Test Pattern Generation for Synchronous Sequential circuits.

The traditional *topological approach* [NiPa91] is based on extending to sequential circuits the branch and bound techniques developed for combinational circuits by adopting Huffman's Iterative Array Model, and the method's effectiveness heavily relies on the heuristics adopted to guide the search. The approach uses a complete algorithm, and therefore it can be exploited to identify untestable and redundant faults. However, thresholds are normally introduced to avoid the exploration of the whole search space, and redundancy identification often fails when applied to large circuits, where the search space is excessively large to be explored.

The recently explored *symbolic approach*[4] exploits techniques which were initially developed for formal verification, based on the extraction and manipulation of the Boolean functions implemented by the circuit. This approach is based on a complete algorithm, too, and is very effective when small- and medium-sized circuits are considered. Being an exact method, it can identify all untestable faults for circuits it is able to deal with. Unfortunately, it is completely inapplicable when dealing with circuits having more than some tens of Flip-Flops. This greatly limits its usefulness in practice.

Finally, the *simulation-based approach*[2] consists of generating pseudo random sequences, fault simulating them, and then modifying their characteristics to increase the obtained fault coverage. Simulation-based approaches can be adopted thanks to the recent advancements of state-of-the-art fault simulation techniques[14].

In the last few years, several methods[20,18,16] have been proposed, which combine the simulation-based approach with the use of Genetic Algorithms[11] (GAs). In fact, Evolutionary Techniques allow to tame randomness and successfully exploit it for optimal solutions finding. Results showed that the approach is very flexible and provides good results for large circuits, where other methods fail.

Due to their characteristics, GA-based ATPGs attracted a lot of interest in the past years: since their first proposal, many improvements have been introduced, and the results reported in the literature show that they are now competitive with the more traditional deterministic approach[7]. However, some resistance to their introduction as commercial products in an industrial environment still exists, mainly due to the non-conventional and not-deterministic nature of the approach they use.

The goal of this paper is to provide the reader with an overview about the state of the art in the field of GA-based ATPGs: GAs are first outlined (Section 2). GATTO is then adopted as a representative tool of the GA-based ATPGs class: GATTO was first proposed[16] and then improved[6]; extensions for parallel systems exploitation[5] and Partial Scan support[8] were also

described. In this paper, the reader is provided with a comprehensive and updated overview of the whole GATTO system: in particular, the GATTO approach is described in Section 3, and the possibilities it offers in terms of experimental results are explained in Section 4. Other GA-based approaches to sequential ATPGs are summarized in Section 5. Some conclusions are finally drawn in Section 6.

2 GENETIC ALGORITHMS

Genetic Algorithms[11] (GAs) have been long investigated as a possible solution for many search and optimization problems.

GAs are *evolutionary algorithms* that mimic the way nature improves the characteristics of living beings. Each solution (*individual*) is represented as a string (*chromosome*) of elements (*genes*); a *fitness value* is assigned to each individual, based on the value given by an *evaluation function*. The evaluation function measures how close the individual is to the optimum solution. A set of individuals constitutes a *population* that evolves from one generation to the next through the creation of new individuals and the deletion of some old ones. The process starts with an initial population created in some way, e.g., randomly. *Evolution* can take two forms:

- *cross-over*: the chromosomes of two individuals are combined to obtain a new individual and inserted in the population to replace another individual, e.g., the one with the lowest fitness (*elitism*). The cross-over is generally backed by techniques ensuring that the selection probability of each individual is proportional to its fitness. Techniques such as *roulette-wheel* or *tournament selection* are often used for this purpose. New individuals are thus likely to have a higher fitness than those they replace. The process is oriented towards those regions of the search space where optimal solutions are supposed to exist.
- *mutation*: a gene of a selected individual is randomly changed. This provides additional chances of entering unexplored regions.

Evolution is stopped when either the goal is reached or a maximum computational effort has been spent.

3 THE GATTO ALGORITHM

We chose the GATTO algorithm as a representative of the GA-based sequential ATPGs class. Other methods belonging to the same class are outlined in Section 7.

3.1 Overview

The test generation process in GATTO is organized in three phases. Each phase manipulates differently a set of sequences, as sketched in Fig. 2.

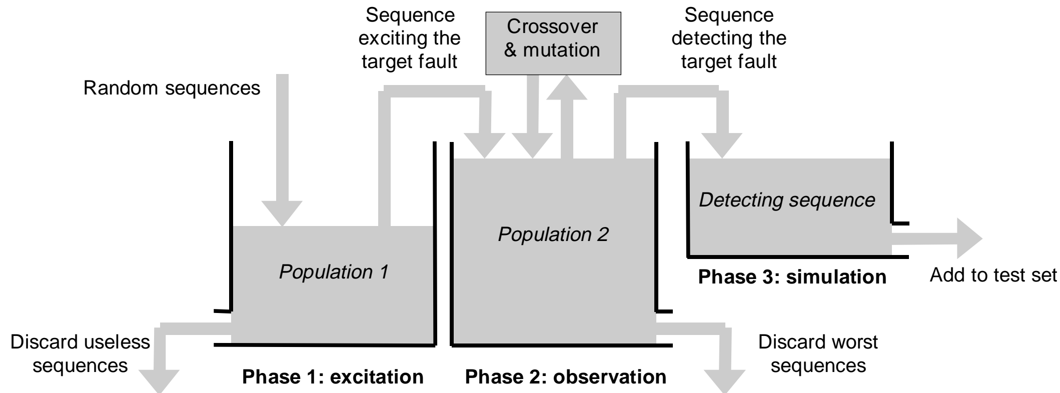

Figure 2: Algorithm organization.

The three phases of the test generation process are detailed in the algorithm's pseudo-code (Fig. 3):

- phase 1: it aims at identifying, among the undetected faults, a fault (`target_fault`) that can be *excited* by some sequence. Circuit activity is stimulated by the application of random sequences;
- phase 2: generation of a test sequence (`new_sequence`), if possible, for the target fault, through the evolution mechanism of a Genetic Algorithm;

- phase 3: fault simulation of the test sequence and search for additionally detected faults (*fault dropping*).

The three phases are repeated until either all the faults have been processed, or a pre-defined maximum number of iterations (MAX_CYCLES) has been reached.

The GATTO algorithm assumes that a single fault-free clock signal feeds all the Flip-Flops, and that a fault-free reset signal is available, so that all Flip-Flops can easily be forced into the all-0 state at the beginning of every sequence.

The remaining of this section describes in higher detail the implementation of the Genetic Algorithm and how the different phases manipulate test sequences.

```
L=Lin;        /* initial length of sequences */
for( i=0; i<MAX_CYCLES; i++ )
{
     /* select a target fault */
     target_fault = phase1 (L);
     if (target_fault == NO_FAULT)
        break;
     /* generate a sequence for the target fault */
     new_sequence = phase2 (target_fault);
     if (new_sequence != NO_SEQUENCE)
     {
        L = length_of(new_sequence);
        /* fault simulate the generated sequence */
        phase3 (new_sequence);
     }
     else
        mark_as_aborted (target_fault);
}
```

Figure 3: Pseudo-Code of the overall algorithm.

3.2 The Genetic Algorithm in GATTO

Generation of a detecting sequence for a fault can be modeled as a search in the space of all the possible sequences applicable to the primary inputs of the circuit. A GA can therefore be used for this purpose, provided that both a suitable encoding for the generic solution, and an effective evaluation function are found.

3.2.1 The Individuals

In GATTO, each *individual* is a single test *sequence*, composed of a variable number of input *vectors* (Fig. 4). Each sequence is to be applied starting from the reset state, one vector per clock cycle. The *population* is a set of NUM_SEQ individuals.

3.2.2 The Fitness

The computation of the fitness value for each individual is based on an intermediate evaluation function. Optimum solutions for the ATPG problem are individuals that detect faults. Through fault simulation, it is trivial to understand whether a given sequence is a solution. Finding an effective *evaluation function* is a much more complex task, since it has to estimate how far an individual is from the optimum.

GATTO estimates the goodness of an individual through the differences it generates in faulty circuits with respect to the fault-free one, assuming that the higher activity a fault produces, the likely it is to be detected. Three heuristic parameters are used for this purpose:

- the weighted number of gates with different values in the fault free and in the faulty circuits. The weight of a gate empirically measures its observability, i.e., the easiness of propagating its value to a primary output or to a Flip-Flop;
- the weighted number of Flip-Flops with different values in the fault free and in the faulty circuits. The weight of a Flip-Flop measures its observability, i.e., the easiness of propagating its value to a primary output;

- the length of the sequence. In order to bias the search toward a compact test set, sequences that generate the same activity with a higher number of vectors are penalized.

The evaluation function adopted in GATTO for a single input vector `v` combines the above heuristic parameters in the following expression:

$$h(v,f) = c_1 * f_1(v,f) + c_2 * f_2(v,f)$$

where `f` is the fault being considered and `v` is an input vector; c_1 and c_2 are normalization constants whose ratio is the ratio of Flip-Flops to gates, while f_1 and f_2 represent the weighted sums mentioned above.

Once the value of `h(v,f)` is computed, via fault simulation, for every vector in a sequence `s`, the evaluation function `H` for the entire sequence `s` is computed according to the best vector it contains:

$$H(s,f) = \max_{v \in s}(\text{LENGTH_HANDICAP}^{(\text{position of } v \text{ in } s)} * h(v,f)).$$

`LENGTH_HANDICAP` is a constant parameter ranging between 0 and 1; thanks to this coefficient, shorter sequences are preferred and the final test length is reduced. The position of a vector in a sequence is its distance (in terms of number of clocks) from the rest state; the first vector has position 0.

The fitness function `F(s,f)` used in the Genetic Algorithm in phase 2 is obtained from `H(s,f)` via *linearization*: the individuals are sorted in decreasing order with respect to `H`, and the value `NUM_SEQ` is assigned to the fitness of the first individual, the value `NUM_SEQ-1` to the second, and so on.

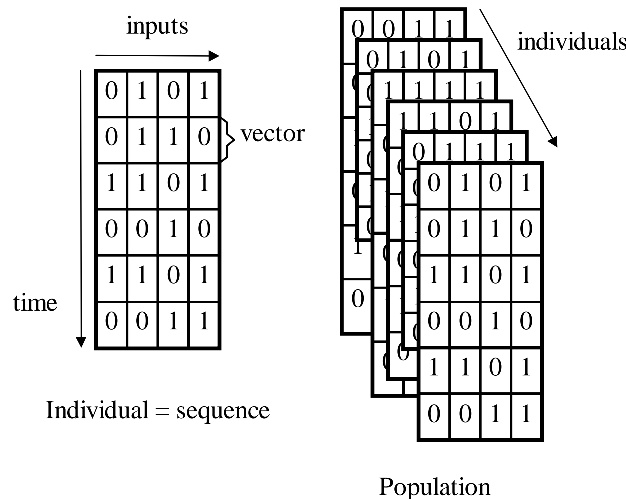

Figure 4: Individual encoding.

3.2.3 The Cross-over Operators

In GATTO, the creation of an offspring is performed thanks to a cross-over operator, that merges the genes of two parent individuals. Two different cross-over operators are currently used in GATTO, and selected on the basis of a random choice:

- a *horizontal*, *two-cut* cross-over (Fig. 5, left): the new sequence is composed of some vectors coming from either parent, according to the position of two randomly generated cut points x_1 and x_2. This operator aims at combining a fault excitation sequence coming from the first parent with a fault observation sequence from the second one. Unfortunately, there is no a priori guarantee that the vectors coming from the second parent produce in the new sequence the same behavior they produced in the parent sequence, as the state from which they are applied is different.
- a *vertical*, *uniform* cross-over (Fig. 5, right): the off-spring does not inherit whole vectors from parents: rather, the values for each input are taken either from one parent or from the other, depending on a random choice. The length of the new sequence is the longest between the two parent ones: inputs taken from the shortest parent are completed with random values.

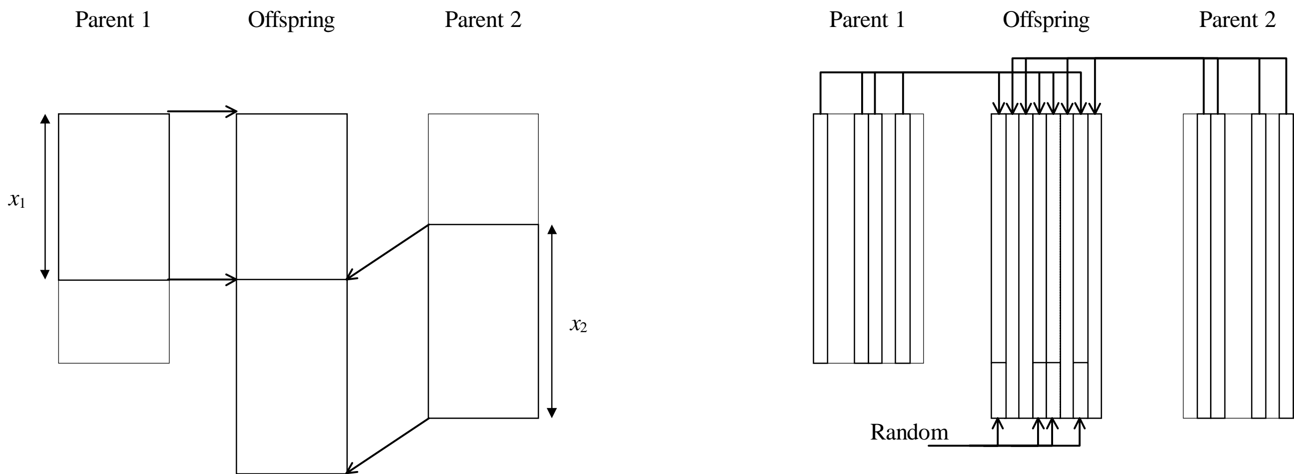

Figure 5: Cross-over operators

3.2.4 The Mutation Operators

The *mutation* mechanism allows to introduce new variety in the population. Mutation is applied, with a probability p_m, on the output of a cross-over operation. Different mutation operators are implemented in GATTO, and selected on the basis of a random choice:

- *single-bit* mutation: it randomly selects a newly generated test sequence and complements a single bit within it;
- *add-vector* mutation: it introduces a randomly generated vector in a random position within the existing sequence; thanks to this operator, longer sequences are generated and evaluated;
- *delete-vector* mutation: it removes a randomly selected vector from the existing sequence: if the vector was not essential, the evaluation function of the sequence can increase.

3.2.5 The Evolution Rules

Given the above definitions for individuals and operators manipulating them, the Genetic Algorithm aims at the evolution of the population, towards regions of the search space where the fitness function is higher.

In order to have chances to generate offspring with higher fitness than their parents, candidates for the cross-over operator are selected on a probabilistic basis: the likelihood of an individual to be selected is proportional to its fitness. The best sequences are therefore more likely to provide vectors for the creation of new individuals. The *roulette-wheel* technique[11] is used for this purpose.

One of the two cross-over operators and, possibly, one of the three mutation operators are then applied, and the fitness function of the new individual is evaluated. At each generation, `NEW_IND` newly created individuals replace the worst individuals in the previous generation. The survival of the best `NUM_SEQ-NEW_IND` individuals from one generation to the next (*elitism*) is thus ensured.

3.3 Phase 1: Fault Excitation

The goal of phase 1 is the identification of a *target fault* by finding some sequence able to excite it. Several random sequences are generated until the evaluation function of a particular fault becomes greater than zero. This fault is then chosen as the *target* one, and phase 2 is entered to generate a test sequence for it.

Phase 1 is based on a pseudo-random search (Fig. 6): sequences composed of `L` vectors are randomly generated in groups of `NUM_SEQ`. Each sequence is fault-simulated with respect to all the not yet detected faults. If no fault produces an evaluation function greater than zero, a new set of sequences is generated, whose length `L` is increased. Otherwise, the fault with the maximum value of the evaluation function is selected as target fault. The process is stopped when a maximum number `MAX_ITER` of iterations is reached. Faults detected during this phase are dropped from the fault list, and the corresponding sequences are inserted in the final set of test sequences.

The initial value of L for the first activation of phase 1 (L_{in}) is computed from the topological characteristics of the circuit. L is then dynamically updated by using for each phase 1 the length of the test sequence generated by the last phase 2.

The initial population, too, is randomly generated the first time phase 1 is activated, while for subsequent runs sequences being generated by the last phase 2 are reused, since they potentially have a higher quality.

The reason why phase 1 is mostly based on a pseudo-random search is that no evaluation function has yet been found, that reliably estimates how far a fault is from being excited.

```
FAULT phase1 (SEQUENCE_LENGTH L)
{     A = {sequences in the population of the last phase 2} ;
      iteration_counter = 0;
      while ( iteration_counter < MAX_ITER )
      {  fault simulate sequences in A ;
         if (some faults are detected)
         {
            add the detecting sequence to the test set;
            drop (detected faults);
         }
         if (some fault f is excited)
            return ( f ); /* select f as target fault */
         A = {NUM_SEQ random sequences of length L};
         iteration_counter ++;
         L = increase(L);
      } /* MAX_ITER */
      return ( NO_FAULT ); /* no target fault found */
}
```

Figure 6: Pseudo-code of phase 1.

```
SEQUENCE phase2 (TARGET_FAULT ft)
{
      the initial population P is inherited from phase 1;
      for (i=0; i<MAX_GEN; i++)
      {
         for (each individual s ∈ P)
            compute the fitness function F(s,ft);
         newP = P;
         for (w=0; w<NEW_IND; w++)
         {
            select two individuals in P; /* roulette wheel */
            apply a cross-over operator, generating a new individual s;
            apply a mutation operator to s with probability pm;
            newP = newP ∪ {s};
         } /* NEW_IND */
         P ={best NUM_SEQ individuals in P};/* update the population */
         for (each individual s ∈ P)
            if (s detects ft)
               return (s);
      } /* MAX_GEN */
      return (NO_SEQUENCE);
}
```

Figure 7: Pseudo-Code of phase 2.

3.4 Phase 2: Fault Observation

Once a target fault has been selected, and a sequence exciting it is found, phase 2 is activated. The goal of phase 2 is to improve the exciting sequence until it becomes a detecting one. This phase is the kernel of the GATTO ATPG, and is based on a Genetic Algorithm whose implementation was detailed in 3.2.

The initial population of phase 2 is composed of the last group of NUM_SEQ sequences generated in phase 1. The target fault f_t only is considered in this phase. A fitness function is associated to each sequence, as described above.

Once a new population has been generated, the evaluation function $H(s,f_t)$ is repeatedly evaluated through fault simulation until one of the following conditions is met (Fig. 7):

- the target fault is detected: the corresponding sequence is then inserted in the final set of test sequences. In this case phase 3 is entered;
- a given maximum number of generations MAX_GEN is generated without detecting the fault: this is then marked as *aborted* and is denied future possibilities of becoming a target fault. The algorithm then returns to phase 1.

3.5 Phase 3: Fault Dropping

Once a detecting sequence has been successfully generated by an activation of phase 2, the target fault f_t is dropped from the fault list. Since phase 2 only concentrates on the target fault, there is a possibility that the same sequence detects some other faults. A fault simulation experiment, aiming at *dropping* additionally detected faults, is therefore the goal of phase 3. By contrast, since in phase 1 all faults are being considered, for faults detected in phase 1 no phase 3 needs to be activated.

Given the assumptions, namely a strictly synchronous sequential circuit and single stuck-at faults, fault simulation poses no conceptual problem, and is amply catered for by many efficient algorithms and tools. In particular, a PROOFS-like fault-parallel event-driven approach[14] has been chosen and implemented in the tool.

4 EXPERIMENTAL RESULTS

In order to verify the effectiveness of Genetic Algorithms when applied to the ATPG problem, we performed a set of experiments on standard benchmark circuits. As a representative of the class of GA-based ATPGs, GATTO is selected to collect experimental data. Results obtained with other algorithms belonging to the same class do not differ in the conclusions that can be drawn.

After presenting some figures concerning the performance of GATTO, we will compare it with a state-of-the-art topological ATPG, namely HITEC[15].

4.1 Experimental Environment

In the experiments a set of sequential benchmark circuits[3] was used, being composed of the ones distributed at the ISCAS'89 conference(the modified versions for circuits s208, s420, and s838 have been used), and the ones known as *Addendum* to the ISCAS'89 benchmark set.

These circuits represent a wide range of small- and medium-sized circuits and are known to have very different characteristics: some of them correspond to the control part of larger circuits, some others also include data paths, some are hand crafted by the designers, while some others are obtained via automated synthesis.

For the purpose of these experiments, we used the GATTO default parameter values (NUM_SEQ=20, MAX_GEN=15, MAX_CYCLES=15). An EDIF netlist is derived from the original benchmark format. The fault list given to GATTO has been generated by the Sunrise commercial fault collapsing tool[22]. It is assumed that all the Flip-Flops in the circuits are resettable, and GATTO generates sequences starting from the all-0 state. The generated vectors are fault simulated with respect to the complete fault list by the Sunrise commercial fault simulator to validate the obtained results in terms of attained Fault Coverage.

Tab. 1 reports the results in terms of Fault Coverage, CPU time and Test Length for the whole set of benchmarks circuits. Experiments have been performed on a Sun SPARCstation 20/50 with a 64 Mbyte memory, running the SunOs operating system. In order to exploit the full power of the fault-parallel fault simulation technique[14], 32 or 64 faults were simulated in parallel exploiting the inherent word parallelism.

4.2 GATTO vs. Topological ATPGs

In order to provide the reader with a sound comparison between the performance of Genetic Algorithm-based ATPGs and the ones of other methods, we ran a fair comparison of GATTO with HITEC, a state-of-the-art topological algorithm described in[15].

A suitable environment[7] has been created to guarantee that the tools run in equivalent conditions: the same circuit and fault list description, detection mechanism, and hardware and software platforms are used.

HITEC has been run with the standard parameter values, and has been forced to use the fault list collapsed as described above. As HITEC does not accept Flip-Flops with the reset signal, we modified the circuit by introducing an AND gate on the data-input of each Flip-Flop and then connecting one input of this gate to an external reset signal. Faults on the added circuitry have not been considered.

Fault Coverage and CPU time have to be considered together, as any increase in the value of the former normally involves an increase in that of the latter, and vice versa. Comparing the CPU time requirements when the attained Fault Coverage is different is thus often a meaningless operation.

Therefore, the approach we adopted to compare the tools is as follows. We first observed which tool produces the best Fault Coverage and CPU time figures for each circuit. Then, those circuits are identified, for which one tool is able both to reach the highest fault coverage, and to require the lowest CPU time. In these cases, the corresponding tool is clearly the *winner* with respect to the other.

The results are graphically reported in Fig. 8. GATTO is the absolute winner in 28 cases out of 42 circuits, while HITEC is the absolute winner in 2 cases out 42. We do not believe that it is possible to comment about the 10 circuits where no tool was the winner, however, we could note that GATTO has a better fault coverage in 7 cases out 10, while HITEC is more effective in providing lower CPU time requirements.

4.3 GATTO vs. Symbolic ATPGs

We performed[7] a fair comparison between the performance of GATTO and that of a symbolic ATPG. As the main result, we showed that the latter is able to outperform GATTO in terms of attained Fault Coverage and required CPU time for most of the small- and medium-sized benchmark circuits. However, no way has been found up to now to extend the applicability of symbolic ATPGs to large circuits (i.e., having more than some tens of Flip-Flops), so that at the moment they do not represent a real solution to the general sequential ATPG problem.

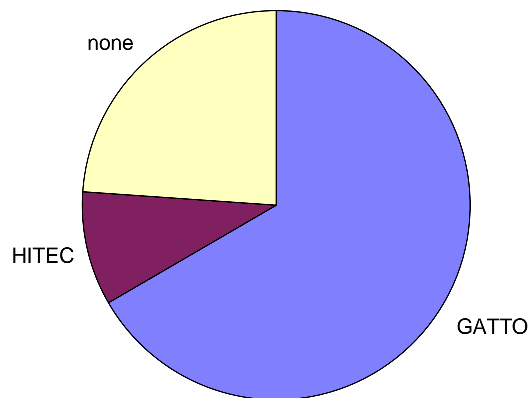

Figure 8: Winning tool after comparisons between GATTO and HITEC.

5 OTHER APPROACHES

Proposals for exploiting Genetic Algorithms for solving the sequential ATPG problem appeared since the beginning of the 90s. All of them benefit from the previous work in the area of simulation-based ATPG techniques[2].

In this section, we first summarize the characteristics of the main GA-based ATPGs so far proposed in the literature. We will then outline two proposals aiming at combining the advantages of GA-based and topological ATPGs, and we finally investigate how parallel and distributed architectures can be exploited to speed-up one of the proposed GA-based ATPGs.

5.1 Basic GA-based ATPGs

Before entering into the details of other ATPG approaches, it is important to underline the fact that they face a slightly different problem than GATTO. In fact, all of them assume that the starting state for all the circuits is the one in which all the Flip Flops hold the unknown value (X), while GATTO assumes the all-0s state as the starting one.

5.1.1 CRIS

In CRIS[20], focus is put on devising an ATPG for gate- and switch-level descriptions able to deal with very large circuits.

The conventional GA structure is adopted, in which populations evolve through mutation and crossover. Each individual is a sequence. The evaluation function is composed of two terms: the former accounts for balanced activity in the circuit, and requires logic simulation only, the latter evaluates how many faults have been detected, and requires fault simulation. Crossover is performed by first evaluating the useful parts in the parents (i.e., those producing activity in some untested parts of the circuit) and then combining the useful parts from the parents.

Results provided by the authors show that CRIS is very fast (being based on logic simulation) and produces compact test sequences; however, it sometimes fails in producing good fault coverage.

5.1.2 GATEST

GATEST[18] is organized in two parts: in the first, single test vectors are generated by the Genetic Algorithm, which are able to increase the value of the already generated test sequence; in the second part, the GA generates test sequences. As a consequence, in the first part individuals correspond to single test vectors, while in the second they are test sequences.

The algorithm goal and evaluation function changes according to the phase:

- in the first phase, the algorithm goal is the initialization of all Flip Flops in the circuit: a population of sequences is thus considered, and every sequence is evaluated through logic simulation only
- in the second phase, all the flip flops are assumed to be initialized, and the goal is the increase of the attained fault coverage by finding new vectors able to detect additional faults
- in the third phase, test vectors are evaluated in terms of their ability to produce good and faulty activity in the circuit.

GATEST uses a modified version of PROOFS[14] to perform Fault Simulation.

5.1.3 DIGATE

DIGATE[12] is organized in three phases:

- the first phase selects a target fault as the one with the maximum activity so far
- the second phase aims at activating the target fault
- the third phase looks at a sequence able to make the target fault observable at the circuit Primary Outputs.

The main innovation in DIGATE are the techniques proposed for speeding-up the third phase: distinguishing sequences are pre-computed, able to propagate a fault effect from a single Flip-Flop to the POs. Every time a new fault is detected, new distinguishing sequences are possibly identified and stored for future use.

5.2 Combined GA-based and topological ATPGs

To overcome some of the limits of the pure GA-based approaches, several researchers proposed hybrid techniques combining the advantages of Genetic Algorithms with those of deterministic ATPGs. We briefly outlines here the two main proposals going in this direction.

5.2.1 Hybrid CRIS

A tool, in which a topological ATPG is combined with CRIS, has been proposed[21].

The main idea is to stop CRIS activities when these are likely to have reached a bottom point in terms of efficiency. This happens for example when CRIS is not able to improve the fault coverage for a given number of generations. In this case, the topological ATPG is activated, which is able either to identify untestable faults (a possible cause of the low efficiency),

or to provide test sequences which are then inserted in the current population. Once the critical point is overcome, the GA is resumed.

Experimental results gathered by the authors show that the hybrid tool is able to significantly outperform HITEC in terms of reached fault coverage and required CPU time.

5.2.2 GA-HITEC

GA-HITEC[19] results from combining together GATEST[17] and HITEC[15]. The topological algorithm implemented in the latter is exploited for fault excitation and propagation, while the Genetic Algorithm implemented in the former performs state justification.

The algorithm cycles through iterations: in each iteration, a fault is first selected. Then, the topological (i.e., branch and bound) algorithm is activated, and generates one vector, able to excite the fault and propagate its effects to a PO or a FF in a single time frame. Next, the fault effects are possibly propagated to a PO in successive time frames. At this point, a Genetic Algorithm is activated to perform state justification. If the GA fails, the topological one is activated again to check whether the fault is untestable.

The fitness function proposed to identify the sequence leading to the required state takes into account the number of flip flops having the correct value in the good and faulty circuits.

Experimental results show that GA-HITEC is able to reach higher fault coverages than HITEC for most circuits, and that it is also faster in many cases. However, it is still less efficient in some cases, due to the time it wastes in looking for test sequences for untestable faults.

6 CONCLUSIONS

The paper provides an overview of ATPG methods based on Genetic Algorithms: GATTO has been used as an example to describe the characteristics of the whole class, which have been compared to the ones of topological and symbolic ATPGs. The algorithms exploited by other GA-based ATPGs are also summarized. Thanks to the improvements devised in the last years, GA-based ATPGs effectively compete with topological ones, and are now ready for being exploited in commercial tools used in real industrial environments; experimental results show that they can provide very good Fault Coverage with limited CPU time requirements, even with large circuits. As further advantages, GA-based ATPGs are versatile enough to be easily ported on parallel and distributed architectures, or to be integrated with existing approaches to exploit the most suitable techniques.

7 ACKNOWLEDGMENTS

The authors wish to thank Dr. Massimo Violante and Dr. Flavio Bianchi for their contribution in the implementation of parts of GATTO and for performing most of the experiments. Special thanks are due to Dr. Giovanni Squillero, who wrote the simulation procedures GATTO is based on. The contribution of Prof. Paolo Prinetto is also acknowledged for the many fruitful discussions.

8 REFERENCES

1. M. Abramovici, M. A. Breuer, A. D. Friedman: *Digital systems testing and testable design*, Computer Science Press, New York, NY (USA), 1990
2. V.D. Agrawal, K.-T. Cheng, P. Agrawal, "CONTEST: A Concurrent Test Generator for Sequential Circuits," Proc. 25th Design Automation Conference, 1988, pp. 84-89
3. Benchmark circuits downloadable at `http://www.cbl.ncsu.edu/www/CBL_Docs/Bench.html`
4. H. Cho, G.D. Hatchel, F. Somenzi, "Redundancy Identification/Removal and Test Generation for Sequential Circuits Using Implicit State Enumeration," *IEEE Trans. on CAD/ICAS*, Vol. CAD-12, No. 7, pp. 935-945, July 1993
5. F. Corno, P. Prinetto, M. Rebaudengo, M. Sonza Reorda, E. Veiluva, "A Portable ATPG Tool for Parallel and Distributed Systems," *Proc. IEEE VLSI Test Symposium*, 1995
6. F. Corno, P. Prinetto, M. Rebaudengo, M. Sonza Reorda, R. Mosca: "Advanced Techniques for GA-based sequential ATPGs," Proc. *IEEE European Design & Test Conf.*, 1996, pp. 75-79
7. F. Corno, P. Prinetto, M. Rebaudengo, M. Sonza Reorda: "Comparing topological, symbolic and GA-based ATPGs: an experimental approach," Proc. *IEEE Int. Test Conf.*, 1996

8. F. Corno, P. Prinetto, M. Rebaudengo, M. Sonza Reorda: “Partial Scan Flip Flop Selection for Simulation-based Sequential ATPGs,” Proc. *IEEE Int. Test Conf.*, 1996, pp. 558-564
9. 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*, August 1996
10. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, V. Sunderam, “PVM 3 User’s Guide and Reference Manual,” Oak Ridge Nat. Lab., Internal Report ORNL/TM-12187, May 1993
11. D.E. Goldberg, “Genetic Algorithms in Search, Optimization, and Machine Learning,” Addison-Wesley, 1989
12. M.S. Hsiao, E.M. Rudnick, J.H. Patel, “Automatic Test Generation Using Genetically-Engineered Distinguishing Sequences,” Proc. *IEEE VLSI Test Symp.*, 1996, pp. 216-223
13. Z. Michalewicz, “Genetic Algorithms + Data Structures = Evolutionary Programs,” Springer Verlag, Berlin (D), 1992
14. T.M. Niermann, W.-T. Cheng, J.H. Patel, “PROOFS: A Fast, Memory-Efficient Sequential Circuit Fault Simulator,” *IEEE Trans. on CAD/ICAS*, Vol. 11, No. 2, pp. 198-207, February 1992
15. T. Niermann, J.H. Patel, “HITEC: A Test Generator Package for Sequential Circuits,” *Proc. European Design Automation Conf.*, 1991, pp. 214-218
16. P. Prinetto, M. Rebaudengo, M. Sonza Reorda: “An Automatic Test Pattern Generator for Large Sequential Circuits based on Genetic Algorithms,”, *Proc. Int. Test Conf.*, 1994, pp. 240-249
17. E.M. Rudnick, J.G. Holm, D.G. Saab, J.H. Patel, “Application of Simple Genetic Algorithms to Sequential Circuit Test Generation,” *Proc. European Design & Test Conf.*, 1994, pp. 40-45
18. E.M. Rudnick, J.H. Patel, G.S. Greenstein, T.M. Niermann, “Sequential Circuit Test Generation in a Genetic Algorithm Framework,” *Proc. Design Automation Conf.*, 1994, pp. 698-704
19. E.M. Rudnick, J.H. Patel, “Combining Deterministic and Genetic Approaches for Sequential Circuit Test Generation,” *Proc. Design Automation Conf.*, 1995, pp. 183-188
20. D.G. Saab, Y.G. Saab, J. Abraham, “CRIS: A Test Cultivation Program for Sequential VLSI Circuits,” *Proc. Int. Conf. on Computer Aided Design*, 1992, pp. 216-219
21. D.G. Saab, Y.G. Saab, J. Abraham, “Iterative [Simulation-Based+Deterministic Techniques]=Complete ATPG,” *Proc. Int. Conf. on Computer Aided Design*, 1994, pp. 40-43
22. Sunrise Reference Manual, *Sunrise Test Systems*, 1995

Circuit	*Fault* Coverage	*CPU Time*	*Test Length*
	%	[s]	#vectors
s208.1	91.20	33.7	1,096
s298	91.30	35.7	302
s344	97.71	27.5	141
s349	97.20	28.2	144
s382	85.97	75.5	840
s386	91.89	52.5	418
s400	84.45	109.5	916
s420.1	55.77	78.9	797
s444	87.13	96.4	1,434
s499	83.93	58.9	465
s510	100.00	32.5	989
s526	76.54	108.0	1,050
s526n	75.18	107.0	862
s635	0.06	190.1	1
s641	88.70	100.8	395
s713	84.83	120.6	557
s820	55.49	166.7	669
s832	58.91	145.0	425
s838.1	41.02	163.7	1,323
s938	41.02	164.9	1,323
s953	99.46	80.5	1,099
s967	98.60	111.2	1,223
s991	97.60	201.2	448
s1196	99.24	198.8	1,805
s1238	95.62	221.7	1,554
s1269	99.88	123.4	450
s1423	93.96	349.1	2,691
s1488	98.44	232.2	1,824
s1494	97.27	179.2	1,244
s1512	58.40	268.8	772
s3271	99.50	693.4	2,529
s3330	78.27	844.6	2,028
s3384	91.38	932.5	888
s4863	97.06	966.3	1,533
s5378	71.73	1,137.6	919
s6669	100.00	159.7	592
s9234	7.84	3,370.3	9
s13207	27.32	6,296.8	544
s15850	8.37	9,427.7	153
s35932	89.85	13,317.5	903
s38417	23.49	40,594.3	1,617
s38584	50.65	35,848.3	8,065

Table 1: GATTO results.