Future Generation Computer Systems 16 (1999) 291–305

FGCS
FUTURE GENERATION COMPUTER SYSTEMS

Generating high-quality random numbers in parallel by cellular automata

Marco Tomassini [a,*], Moshe Sipper [b], Mosé Zolla [a], Mathieu Perrenoud [a]
[a] Institute of Computer Science, University of Lausanne, 1015 Lausanne, Switzerland
[b] Logic Systems Laboratory, Swiss Federal Institute of Technology, IN-Ecublens, CH-1015 Lausanne, Switzerland

Accepted 17 March 1999

Abstract

Many important computer simulation methods rely on random numbers, including Monte Carlo techniques, Brownian dynamics, and stochastic optimization methods such as simulated annealing. Several deterministic algorithms for producing random numbers have been proposed to date. In this paper we concentrate on generating pseudo-random sequences by using cellular automata, which offer a number of advantages over other methods, especially where hardware implementation is concerned. We study both hand-designed random number generators as well as ones produced by artificial evolution. Applying an extensive suite of tests we demonstrate that cellular automata can be used to rapidly produce high-quality random number sequences. Such automata can be efficiently implemented in hardware, can be used in such applications as VLSI built-in self-test, and can be applied in the field of parallel computation. ©1999 Elsevier Science B.V. All rights reserved.

Keywords: Cellular automata; Random number generators; Evolutionary Computing; Parallel computation

1. Introduction

Several important computer simulation methods rely on random numbers, including Monte Carlo techniques, Brownian dynamics, and stochastic optimization methods such as simulated annealing. The quality of the results of these methods critically depends on the quality of the random sequence as measured by suitable statistical tests. Computational efficiency is also an important aspect when very long sequences of random numbers have to be produced. Random numbers are also needed in another important application area: built-in self-test devices for VLSI circuits. In this case, as well as for fine-grained massively parallel computers and for on-board applications, it is essential that the random number generator also be amenable to hardware implementation in terms of area, number of gates, and complexity of the interconnections.

Several deterministic algorithms for producing random numbers have been proposed to date. A short review of the principal pseudo-random number generator methods (RNG) is given in Section 2. In the present study we concentrate on generating pseudo-random sequences by using cellular automata (CA), a short account of which is given in Section 3. Cellular automata offer a number of advantages over other methods, especially where hardware implementation is concerned. Among the beneficial features of CA for VLSI implementation one can cite simplicity, regularity, and locality of interconnections. Cellular automata for random number generation can be constructed by hand essentially by looking at the structure

[*] Corresponding author. Tel.: +41-21-692-3589; fax: +41-21-692-3585
E-mail address: marco.tomassini@iismail.unil.ch (M. Tomassini)

of the bit patterns generated over time, with some theoretical results offering guidance. Another possibility is to let the CA generator be artificially evolved by a genetic algorithm. Here a measure of adequacy or *fitness* is used in order to drive the population of candidate generators towards better and better performance over time (see Section 4). A first step in the latter direction was taken in [12]. In this work it was demonstrated that good CA-based random number generators could be evolved. The present work confirms and extends these results by using much longer pseudo-random sequences and by subjecting them to a more stringent and elaborate set of randomness tests (discussed in Section 5). Moreover, we have carried out many more evolutionary simulations, strengthening our confidence in the repeatability and overall stability of the dynamics of the evolutionary process. Our results, presented in Section 6, show that CA-based RNGs can yield long-period, high-quality random number sequences, possibly rivaling those obtained by well-known congruential generators. We conclude that with their advantageous hardware properties, CA random number generators offer a realistic alternative to other methods; this is especially true in the case of VLSI implementation, fine-grained massively parallel machines for statistical physics simulation, and built-in self-test circuits.

2. An overview of random number generators

Random numbers are needed in a variety of applications, yet finding good random number generators is a difficult task [10]. All practical methods for obtaining random numbers are based on deterministic algorithms, which is why such numbers are more appropriately called *pseudo*-random, as distinguished from true random numbers resulting from some natural physical process. In the following we will limit ourselves to *uniformly distributed* sequences of pseudo-random numbers; however, there are well-known ways for obtaining sequences distributed according to a different distribution starting from uniformly distributed ones.

Random number generators must possess a number of properties if they are to be useful in lengthy stochastic simulations such as those used in computational physics. The most important properties from this point of view are good results on standard statistical tests of randomness, computational efficiency, a long period, and reproducibility of the sequence.

There exist many ways for generating random numbers on a computer, the most popular one being the *linear congruential generators*. Linear congruential generators are based on the following recurrent formula:

$$X_{n+1} = (aX_n + c) \bmod m, \quad n \geq 0, \quad m > 0, \quad 0 < a < m.$$

The value $m > 0$ is called the *modulus*, a is the *multiplier*, and c is an additive constant. Ref. [4] describes in great detail how to pick suitable values for these parameters. The sequence clearly has a maximum possible *period* of m, after which it starts to repeat itself. The linear congruential generators are very popular among researchers and most mathematical software packages include one (or more).

So-called *lagged-Fibonacci generators* are also widely used. They are of the form:

$$X_n = (X_{n-r} \text{ op } X_{n-p}) \bmod m.$$

The numbers r and p are called *lags* and there are methods for choosing them appropriately (see [4]). The operator op can be one of the following binary operators: addition, subtraction, multiplication, or exclusive or.

However, it should be noted that from the point of view of hardware implementation both congruential and lagged-Fibonacci RNGs are not very suitable: they are inefficient in terms of silicon area and time when applied to fine-grained massively parallel machines, for built-in self-test, or for other on-board applications.

A third widespread type of generator is the so-called *linear feedback shift register* (LFSR) generators. A pseudo-random sequence is generated by the linear recursion equation

$$X_n = (c_1X_{n-1} + c_2X_{n-2} + \cdots + c_kX_{n-k}) \bmod 2.$$

Linear feedback shift registers are popular generators among physicists and computer engineers. There exist forms of LFSR that are suitable for hardware implementation. However, it turns out that, when compared with equivalent cellular automata-based generators, they are of lesser quality; furthermore, they are less favorable in terms of connectivity and delay, although the area needed by a CA cell is slightly larger than that of an LFSR cell [3]. This is so because a LFSR with a large number of memory elements and feedback has an irregular interconnection structure which makes it more difficult to modularize in VLSI. Moreover, different sequences generated by the same CA are much less correlated than the analogous sequences generated by a LFSR. This means that CA-generated bit sequences can be used in parallel, which offers clear advantages in applications. Our measurements (see Section 6) confirm this view.

3. Cellular automata for random number generation

Cellular automata (CA) are dynamical systems in which space and time are discrete. A cellular automaton consists of an array of cells, each of which can be in one of a finite number of possible states, updated synchronously in discrete time steps, according to a local, identical interaction rule. Here we will only consider Boolean automata for which the cellular state $s \in \{0, 1\}$. The state of a cell at the next time step is determined by the current states of a surrounding neighborhood of cells. The cellular array (grid) is d-dimensional, where $d = 1, 2, 3$ is used in practice; in this paper we shall concentrate on $d = 1$, i.e., one-dimensional grids. The identical rule contained in each cell is essentially a finite state machine, usually specified in the form of a rule table (also known as the transition function), with an entry for every possible neighborhood configuration of states. The cellular neighborhood of a cell consists of itself and of the surrounding (adjacent) cells. For one-dimensional CAs, a cell is connected to r local neighbors (cells) on either side where r is referred to as the radius (thus, each cell has $2r + 1$ neighbors). For two-dimensional CAs, two types of cellular neighborhoods are usually considered: five cells, consisting of the cell along with its four immediate nondiagonal neighbors (also known as the von Neumann neighborhood), and nine cells, consisting of the cell along with its eight surrounding neighbors (also known as the Moore neighborhood). When considering a finite-sized grid, spatially periodic boundary conditions are frequently applied, resulting in a circular grid for the one-dimensional case, and a toroidal one for the two-dimensional case.

Non-uniform (also know as inhomogenous) cellular automata function in the same way as uniform ones, the only difference being in the cellular rules that need not be identical for all cells. Note that non-uniform CAs share the basic “attractive” properties of uniform ones (simplicity, parallelism, locality). From a hardware point of view we observe that there is a slight loss in homogeneity although the resources required by non-uniform CAs are identical to those of uniform ones since a cell in both cases contains a rule.

A common method of examining the behavior of one-dimensional CAs is to display a two-dimensional space-time diagram, where the horizontal axis depicts the configuration at a certain time t and the vertical axis depicts successive time steps (e.g., Fig. 1). The term ‘configuration’ refers to an assignment of ones and zeros at a given time step (i.e., a horizontal line in the diagram).

Using one-dimensional, two-state CAs as a source of random bit sequences was first suggested by Wolfram [14]. In particular, he extensively studied rule 30. (Rule numbers are given in accordance with Wolfram’s convention. [1]) In Boolean form rule 30 can be written as

$$s_i(t+1) = s_{i-1}(t) \;\; XOR \;\; (s_i(t) \;\; OR \;\; s_{i+1}(t)),$$

where the radius is $r = 1$ and $s_i(t)$ is the state of cell i at time t. The formula gives the state of cell i at time step $t + 1$ as a Boolean function of the states of the neighboring cells at time t. Random bit sequences are obtained by sampling the values that a particular cell (usually the central one) attains as a function of time. Often, in order to further decorrelate bit sequences, *time spacing* and *site spacing* are used. Time spacing means that not all the bits generated are considered as part of the random sequence. For instance, one might keep only one bit out of two, referred to as a time space value of 1, which means that sequences will be generated at half the maximal rate. Obviously, other values of the time interval can be used with an increasing loss of clock cycles. In site spacing, one considers only certain sites in a row, where an integer

[1] Wolfram’s numbering scheme for one-dimensional, $r = 1$ rules represents in decimal format the binary number encoding the rule table. For example, $f(1\ 1\ 1) = 1$, $f(1\ 1\ 0) = 0$, $f(1\ 0\ 1) = 1$, $f(1\ 0\ 0) = 1$, $f(0\ 1\ 1) = 1$, $f(0\ 1\ 0) = 0$, $f(0\ 0\ 1) = 0$, $f(0\ 0\ 0) = 0$, is denoted rule 184.

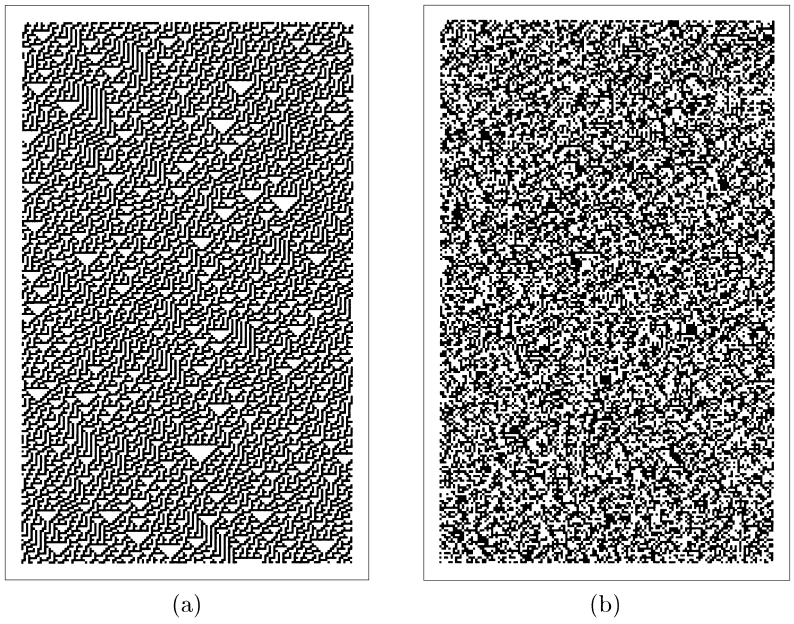

Fig. 1. A one-dimensional random number generator: CA rule 30. Grid size is $N = 150$, radius is $r = 1$. White squares represent cells in state 0, black squares represent cells in state 1. The pattern of configurations is shown through time (which increases down the page). The initial configurations were generated by randomly setting the state of each grid cell to 0 or 1 with uniform probability. (a) No time spacing. (b) Time spacing of 2.

number indicates how many sites are to be ignored between two successive cells. In practice, a site spacing of one or two is common, which means that half or two thirds of the output bits are lost. Fig. 1 demonstrates the workings of a rule-30 CA, both with and without time spacing.

A non-uniform CA randomizer was presented in Ref. [2,3], consisting of two rules, 90 and 150, arranged in a specific order in the grid. In Boolean form rule 90 can be written as

$$s_i(t+1) = s_{i-1}(t)\ \ XOR\ \ s_{i+1}(t),$$

and rule 150 can be written as:

$$s_i(t+1) = s_{i-1}(t)\ \ XOR\ \ s_i(t)\ \ XOR\ \ s_{i+1}(t).$$

The performance of this hybrid (non-uniform) CA in terms of random number generation was found to be superior to that of rule 30. It is interesting in that it combines two rules, both of which are simple linear rules, to form an efficient, high-performance generator. An example application of such CA randomizers was demonstrated in Ref. [1], which presented the design of a low-cost, high-capacity associative memory. Sipper and Tomassini [12] showed that good non-uniform CA randomizers can be evolved, rather than being designed; this evolutionary approach is described in the next section.

4. Evolving cellular automata: cellular programming

The idea of applying the biological principle of natural evolution to artificial systems, introduced more than three decades ago, has seen impressive growth in the past few years. Among artificial evolutionary techniques, *genetic algorithms* are very popular nowadays. A genetic algorithm [9] is an iterative procedure that consists of a constant-size population of individuals, each one represented by a finite string of symbols, known as the *genome*, encoding a possible solution in a given problem space. This space, referred to as the *search space*, comprises all possible solutions to the problem at hand. The algorithm sets out with an initial population of individuals that is generated at random or heuristically. Every evolutionary step, known as a *generation*, the individuals in the current population are *decoded* and *evaluated* according to some predefined quality criterion, referred to as the *fitness*, or *fitness function*. To form a new population (the next generation), individuals are *selected* according to their fitness, and then transformed via genetically inspired operators, of which the most well known are *crossover* and *mutation*. Iterating this procedure, the genetic algorithm may eventually find an acceptable solution, i.e., one with high fitness.

We study one-dimensional, 2-state, $r=1$ non-uniform CAs, in which each cell may contain a different rule. Spatially periodic boundary conditions are used, resulting in a circular grid. A cell's rule table is encoded as a bit string (the "genome"), containing the next-state (output) bits for all possible neighborhood configurations, listed in lexicographic order; e.g., for CAs with $r = 1$, the genome consists of 8 bits, where the bit at position 0 is the state to which neighborhood configuration 000 is mapped to and so on until bit 7, corresponding to neighborhood configuration 111. An evolutionary approach for obtaining random number generators was taken by Koza [5], who applied genetic programming to the evolution of a symbolic LISP expression that acts as a rule for a uniform CA (i.e., the expression is inserted into each CA cell, thereby comprising the function according to which the cell's next state is computed). He demonstrated evolved expressions that are equivalent to Wolfram's rule 30. The fitness measure used by Koza is the *entropy* E_h: let k be the number of possible values per sequence position (in our case CA states) and h a subsequence length. E_h (measured in bits) for the set of k^h probabilities of the k^h possible subsequences of length h is given by:

$$E_h = -\sum_{j=1}^{k^h} p_{h_j} \log_2 p_{h_j},$$

where $h_1, h_2, \dots, h_{k^h}$ are all the possible subsequences of length h (by convention, $\log_2 0 = 0$ when computing entropy). The entropy attains its maximal value when the probabilities of all k^h possible subsequences of length h are equal to $1/k^h$; in our case $k = 2$ and the maximal entropy is $E_h = h$.

Rather than employ a *population* of evolving, uniform CAs, as with genetic algorithm approaches, our algorithm involves a *single*, non-uniform CA of size N, with cell rules initialized at random [11]. Initial configurations are then randomly generated and for each one the CA is run for $M = 4096$ time steps.[2] Each cell's *fitness*, f_i, is accumulated over $C = 300$ initial configurations, where a single run's score equals the entropy E_h of the temporal sequence of cell i. Note that we do not restrict ourselves to one designated cell, but consider all grid cells, thus obtaining N random sequences in parallel, rather than a single one. After every C configurations evolution of rules occurs by applying the genetic operators of crossover and mutation in a completely *local* manner, driven by $nf_i(c)$, the number of fitter neighbors of cell i after c configurations. The pseudo-code of our algorithm is delineated in Fig. 2. Crossover between two rules is performed by selecting at random (with uniform probability) a single crossover point and creating a new rule by combining the first rule's bit string before the crossover point with the second rule's bit string from this point onward. Mutation is applied to the bit string of a rule with probability 0.001 per bit.

There are two main differences between our evolutionary algorithm and a standard genetic algorithm: (a) A standard genetic algorithm involves a population of evolving, uniform CAs; all CAs are *ranked* according to fitness, with crossover occurring between *any* two CA rules. Thus, while the CA runs in accordance with a local rule, evolution proceeds in a *global* manner. In contrast, our algorithm proceeds *locally* in the sense that each cell has access only to its locale, not only during the run but also during the evolutionary phase, and no global fitness ranking is performed. (b) The standard genetic algorithm involves a population of *independent* problem solutions; each CA is run independently, after which genetic operators are applied to produce a new population. In contrast, our CA *coevolves* since each cell's fitness depends upon its evolving neighbors. A thorough examination of cellular programming is provided by Sipper [11] and a shorter review can be found in Ref. [6].

[2] A standard, 48-bit, linear congruential algorithm proved sufficient for the generation of random initial configurations.

```
for each cell $i$ in CA do in parallel
    initialize rule table of cell $i$
    $f_i = 0$ { fitness value }
end parallel for
$c = 0$ { initial configurations counter }
while not done do
    generate a random initial configuration
    run CA on initial configuration for $M$ time steps
    for each cell $i$ do in parallel
        $f_i = f_i +$ entropy $E_h$ of the temporal sequence of cell $i$
    end parallel for
    $c = c + 1$
    if $c$ mod $C = 0$ then { evolve every $C$ configurations}
        for each cell $i$ do in parallel
            compute $nf_i(c)$ { number of fitter neighbors }
            if $nf_i(c) = 0$ then rule $i$ is left unchanged
            else if $nf_i(c) = 1$ then replace rule $i$ with the fitter neighboring rule,
                                 followed by mutation
            else if $nf_i(c) = 2$ then replace rule $i$ with the crossover of the two fitter
                                 neighboring rules, followed by mutation
            else if $nf_i(c) > 2$ then replace rule $i$ with the crossover of two randomly
                                 chosen fitter neighboring rules, followed by mutation
                                 (this case can occur if the cellular radius, $r$, $> 1$)
            end if
            $f_i = 0$
        end parallel for
    end if
end while
```

Fig. 2. Pseudo-code of the cellular programming algorithm

5. Testing random number generators

When generating a sequence of random numbers one often tends to forget that these numbers are not truly random (whatever this may mean), but only pseudo-random. Once we admit the inevitability of this assertion, we would still like to obtain sequences that behave as if they were random. But who is to decide if the numbers are random enough? Statistical theory helps us in this respect. If the sequence passes a number of quantitative statistical tests for randomness then we can affirm with some confidence that the sequence is random at least for practical purposes. Of course, no amount of statistical testing can guarantee that a given sequence is truly random since one can always find a new test that the sequence fails to pass. A classical exposition of statistical tests for randomness is Ref. [4], which has been recently updated.

Our aims in this work are the following:

- to test the performance of some well-known CA RNGs;
- to test the performance of CAs found by artificial evolution;
- to compare the performance of CA-based RNGs with those of some good standard generators;
- and, to study the correlation between different sequences generated from the same CA or from different CAs.

For the tests we have used the Diehard [8] battery of tests developed by Marsaglia [7]. A visual test has also been used as a quick way of discarding obviously unsuitable CA generators. The chi-square test and the correlation factor have been used to study the correlation between sequences. Since the chi-square test is limited to discrete distributions, it is often complemented by the Kolmogorov–Smirnof (KS) test in the continuous case, such as random numbers uniformly distributed between 0 and 1. The chi-square, Kolmogorov–Smirnov, and correlation tests are standard statistical tests, the full explanation of which can be found in [4].

5.1. Cross-correlation

This test is used to detect the existence of correlations between sequences generated from the same CA. It is interesting because if there are no correlations, we can use different sequences from the same CA for parallel stochastic calculations. We have studied the cross-correlation between sequences generated by the same CA with different initial conditions. In other words, the same automaton is initialized with two different random configurations of states and two sequences of the same length and equal time spacing are thus obtained. The correlation between the two sequences is then measured. A second correlation measure involves the extraction of sequences from the same automaton with a given time spacing parameter. We did this with a time spacing value of 2 (see Section 3), which means that the bits of time steps $0, 3, 6, \ldots$ belong to the first sequence while those of time steps $1, 4, 7, \ldots$ belong to the second one. The latter sequence sampling is obviously more favorable in terms of parallel random bit generation since it only makes use of a single CA, while the first one would be preferable in a multi-processor machine.

5.2. The test suite

Diehard is a battery of tests that analyzes 17 different properties of RNGs. The tests are only briefly described herein since the corresponding programs and documentation are freely available on the web [8]. Although there do exist a few other test suites for assessing the randomness of a sequence, the Diehard battery of tests is recognized worldwide as being the most exhaustive.

The overlapping 5-permutation test. Consider a sample composed of one million 32-bit integers. Each series of five consecutive integers could be in one of 120 possible states. After collecting many thousands of state transitions, cumulative counts are made of the number of occurrences of each state and compared with the theoretical distribution of the frequencies.

Binary rank tests for $n \times m$ matrices. These are three tests depending upon the number of integers tested (n) and the corresponding number of bits (m). Matrices are constructed in various ways from the given sequences and the rank of each matrix is calculated for a large number of cases. The frequencies are placed in different categories and then a chi-square test is performed between the rank's empirical matrix and the rank's theoretical matrix frequencies.

Count the ones test. Consider a sequence of bytes. Each byte could contain between zero and eight ones. Theoretical frequencies for the different categories are 1,8,28,56,70,56,28,8,1 (the total sum of which is 256). We group bytes containing zero to two ones in one category, and bytes containing six to eight ones in another one. The new theoretical distribution becomes 37,56,70,56,37. We call the different categories "letters", from A to E. There are 5^5 possible five-letter words, and from a string of 256,000 (overlapping) five letters words, counts are made on the frequencies of each word. The quadratic form in the weak inverse of the covariance matrix of cell counts provides a chi-square test.

Squeeze test. Random integers are transformed into floating-point numbers in the range $[0, 1)$. Consider $k = 2^{31}$ and the function $k = ceiling(k \times U)$, where U is a number in our sequence; j is the number of iterations necessary to reduce k to one. The value of j is measured 100,000 times and the frequencies are then compared to the theoretical ones through a chi-square test.

Missing words tests. The structure of these tests is the following: we consider a string of 32-bit integers. The alphabet is composed of all different combinations of zeros and ones in the sequence of bits. For example, a two-bit alphabet is composed of four letters ((0,0),(0,1),(1,0),(1,1)). We consider a word as being a string composed of n letters, and we traverse the entire sequence as a series of words (with overlapping). The test counts the words that do not appear in the sequence and compares the result with the theoretical distribution for the missing words. If the value for a missing word is too far from the mean of the theoretical distribution we will consider that the series is not random. In these tests the string is always 2^{21} words long. The tests are:

- *Bitstream test.* The file under test is viewed as a stream of bits. Consider an alphabet with two letters 0 and 1; this is called a one-bit alphabet (0,1). The words are 20 letters long (with overlapping). The theoretical distribution of missing words is normally distributed with mean 141,909 and standard deviation equal to 428.
- *OPSO (Overlapping-Pairs-Sparse-Occupancy) test.* A 10-bit alphabet (1024 letters) is used. The words are two letters long (with overlapping). The theoretical distribution of missing words is normally distributed with mean 141,909 and standard deviation equal to 290.
- *OQSO (Overlapping-Quadruples-Sparse-Occupancy) test.* A 5-bit alphabet (32 letters) is used. The words are four letters long (with overlapping). The theoretical distribution of missing words is normally distributed with mean 141,909 and standard deviation equal to 295.
- *DNA test.* A 2-bit alphabet is employed here. The words are 10 letters long (with overlapping). The theoretical distribution of missing words is normally distributed with mean 141,909 and standard deviation equal to 339.

Birthday spacing test. Choose m birthdays in a year of n days. Count the number of days between two birthdays and construct a class for each value of such birthday intervals. The distribution of intervals should be asymptotically that of Poisson with mean $m^3/4n$. In this test we use $n = 2^{24}$ or $n = 2^{18}$ and thus the mean equals 2. Each sample is composed of 500 intervals, each one giving a p-value. We use the nine p-values to do the Kolmogorov–Smirnof test.

Parking lot test. Imagine a square parking of 100×100 cells of side 1 where a car, is represented by a circle of radius 1. Then park randomly each successive car trying to avoid bumping into already parked cars. The number of attempts versus the number of successfully parked cars should be random. After 12,000 attempts the theory shows that the number of successes is normally distributed with mean $= 3523$ and a standard deviation $= 21.9$. The values obtained are compared with the theoretical distribution, and the distance of the experimental one from the mean gives us the p-value. The set of ten p-values thus obtained is used to perform a KS test.

Minimum distance test. One chooses 8000 points in a square of side $= 10,000$ units. Find d, the minimum distance between the $(n^2 - n)/2$ pairs of points. If the points are independent then d^2 has to be exponentially distributed with mean = 0.995. Transformed by $1 - \exp(-d^2/0.995)$,d should be uniformly distributed in the range $[0, 1)$. A KS test on the 100 values obtained serves as a test for the uniformity of the points in the square. The test is repeated 100 times on 8000 different sets of points.

3D sphere test. Choose 4000 random points in a cube of side 1000 units. At each point center a sphere large enough to reach the next closest point. The radius cubed ($radius^3$) of the smallest sphere is exponentially distributed with mean $= 30$. The test generates 4000 spheres 20 times. Each minimum $radius^3$ leads to a uniform variable in the range [0, 1) by applying the transformation $1 - \exp(-r^3/30)$. A Kolmogorov–Smirnof test is done on the 20 p-values.

Overlapping sum test. Random integers are transformed into floating-point numbers in the range $[0, 1)$, thus giving a new series $u_1, u_2, \ldots$. Then, the overlapping sums $s_1 = u_1 + \cdots + u_{100}, u_2 + \cdots + u_{101}, \cdots$ are formed. The sums are normal with a certain covariance matrix. A linear transformation of the sums converts them to a sequence of independent standard normals, which are converted to uniform variables for a KS test. The p-values resulting from the ten KS tests undergo another KS test.

Runs test. This test analyzes the trend of a sequence of numbers. The random integers are transformed into floating-point numbers in the range $[0, 1)$. We want to control the randomness of ascending and descending sub-sequences. The runs-up and runs-down covariance matrices lead to a chi-square test for quadratic forms using the weak inverse of the matrices. Runs are counted for sequences of length 10,000. This test is done ten times, then repeated. It is done both for the up and down series.

Craps test. One plays 200,000 games of craps, finds the number of wins and the number of throws necessary to end a game. The integers are first transformed into floating-point numbers in the range $[0, 1)$, then they are multiplied by 6 and 1 is added to the integer part of the result, giving the value for a throw of a die. The number of wins should exhibit a normal distribution with mean 110,556 and standard deviation 95.6962. To test the number of throws necessary to win, a chi-square test is done. The number of throws giving values larger than 21 are grouped in one category. These values are compared to the theoretical ones and a chi-square test is done.

According to Marsaglia [8], a generator fails to pass a test if the p-value scores are very close to zero or to one, to six or more decimal places. (Thus a $p < 0.025$ or $p > 0.975$ should not be interpreted, as is customary in statistics, as having failed the test at the 0.05 confidence level.)

The visual test. This test permits a rough selection among all conceivable CAs of radius $r = 1$, both uniform and non-uniform. One looks at the space-time diagram (see Section 3) of the CA and if one detects symmetries, triangles, or something else possessing a structure then one can be sure that the CA is not a good RNG. The opposite is false: random-looking CAs could give bad results as far as RNG is concerned. For instance, Fig. 1(a) shows the dynamical evolution of CA rule 30. Although this rule does not provide a very good RNG, it is already random enough so as to pass a number of Diehard tests. Note that Fig. 1(b), in which a time spacing of 2 is employed, seems visually better – a fact that is confirmed by the tests.

Cross-correlation tests. The random number sequences are coded in hexadecimal form. We used two tests: the correlation coefficient and the uniformity of the distribution of pairs of sequence values. The classical method is to calculate the correlation coefficient R, i.e., the correlation between a variable x and $y = f(x)$ in a linear regression. We consider two sequences of the same length l. The first sequence contains the x values and the second one contains the y values. If there is no correlation between these two sequences, the R value will be very close to zero, and lR^2 has to be less than 4 in order to attain a 95% probability that the sequences are not first-order correlated. To improve the quality of our results we have added a new test. Consider the sequences $S_1 = S_1(1), S_1(2), \ldots$ and $S_2 = S_2(1), S_2(2), \ldots$. We create a new sequence by pairing corresponding values of the two sequences: $(S_1, S_2) = (S_1(1), S_2(1)), (S_1(2), S_2(2)), \ldots$. If the sequences are random, the frequencies of the different pairs (0,0),(0,1)...(E,F),(F,F) are equal to $l/256$ where l is the number of values contained in the sequence. A chi-square test measures the differences between the empirical frequencies and the theoretical ones, and the result must be close to 255. The more the value deviates from 255, the worse is the result. A measure of this distance is given by the standard deviation. If the chi-square value (v-value) does not fall between 209.834 and 300.166 (at a 95% confidence interval) it means that the sum of the deviations is not caused by randomness in the data. These two tests were performed on files of 80MB. We have tested the eight sub-sequences of 10MB, the two sub-sequences of 40MB, and the entire 80MB sequence.

6. Results

6.1. Evolution of RNG cellular automata

In order to evolve RNG CAs we applied the cellular programming evolutionary algorithm of Section 4. We observed that in over 100 experiments, four rules tended to dominate the final evolved grids: 90, 105, 150, and 165; interestingly, rule 30 had never emerged. In some cases a non-uniform grid was obtained, while other simulations saw the appearance of uniform CAs. Fig. 3 shows the workings of rule 105.

Previously, Sipper and Tomassini [12] had evolved a 50-cell CA with a melange of rules 90, 150, and 165, and applied a small number of randomness tests to it. Herein, we verified this CA using our extended battery of tests. Furthermore, we constructed a novel CA – demonstrated in Fig. 4 – by building upon our observations of the evolutionary process: each cell of the 50-cell CA is assigned one of the four “good” rules at random – 90, 105, 150, or 165. (These two CAs are referred to henceforth as Sipper and Tomassini, and Tomassini et al., respectively.)

6.2. Results of randomness tests

Classical RNGs. Before examining CA RNGs let us take a look at two classical, well-known ones. The first is a linear congruential RNG, known as CGL, which was tested by Vattulainen [13]. Its formula is the following: $X_{i+1} = (16807\ X_i) \bmod (2^{31} - 1)$. This RNG was employed on IBM computers and it is available in different commercial software packages (e.g., IMSL and MATLAB). Considered good in the past, new empirical tests have revealed it to be less so. The second RNG, known as RAN3, is

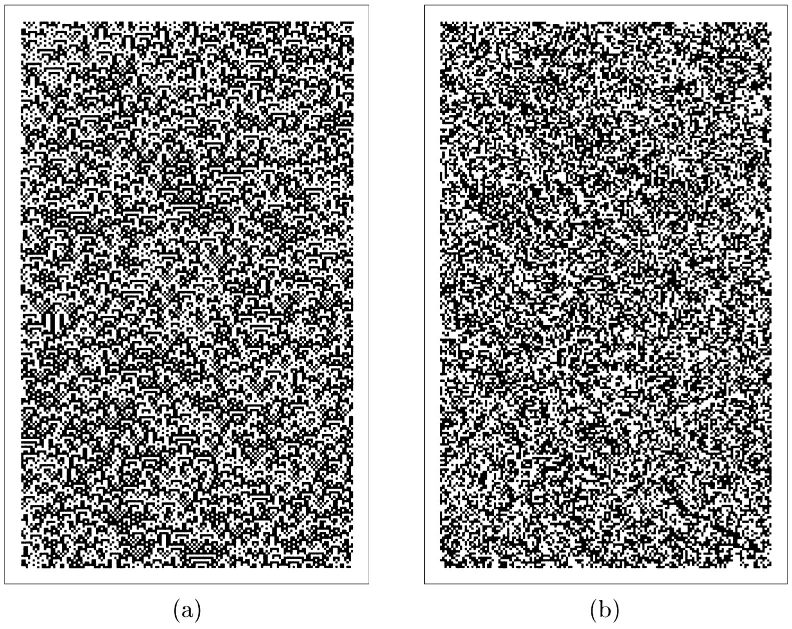

Fig. 3. A uniform one-dimensional random number generator: CA rule 105. (a) No time spacing. (b) Time spacing of 2.

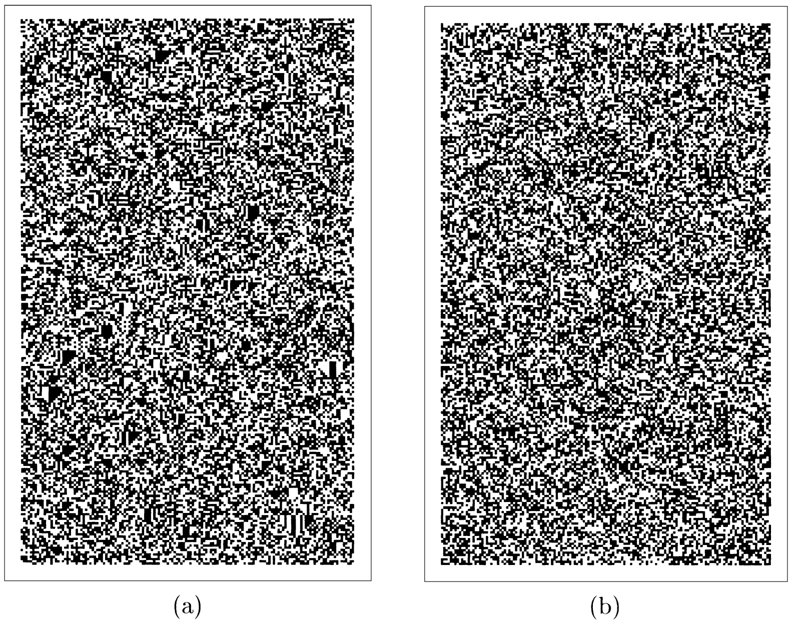

Fig. 4. A non-uniform one-dimensional random number generator: Tomassini et al. (a) No time spacing. (b) Time spacing of 2. Note that there is no marked visible difference between the two versions, though the tests prove otherwise.

Table 1
Diehard p-value results for the CGL and RAN3 random number generators. A generator fails to pass a test if the p-value scores are very close to zero or to one, to six or more decimal places (see text)

Test name	GGL	RAN3
Birthday spacing	0.998	1.000
Overlapping permutation 1	0.422	0.578
Overlapping permutation 2	0.951	0.047
Binary rank 31*31	0.368	0.840
Binary rank 32*32	1.000	0.958
Binary rank 6*8	0.926	0.001
Count the ones	1.000	0.542
Parking lot	0.510	0.204
Minimum distance	1.000	0.911
3D sphere	0.546	0.663
Squeeze	0.228	0.579
Overlapping sum	0.849	0.722
Run up 1	0.838	0.123
Run up 2	0.003	0.638
Run down 1	0.504	0.057
Run down 2	0.886	0.045
Craps number of throws	0.219	0.281
Craps number of wins	0.481	0.250

of the lagged Fibonacci type. Its formula is: $X_i = (X_{i-55} - X_{i-32}) \mod 10^9$. It has a period of $2^{55} - 1$, and 55 numbers (the seeds) are necessary in order to initialize it. This generator was tested by Vattulainen [13], and is considered rather good, though there exist nonetheless some correlations between bits. Table 1 shows the test results for these two RNGs.

Uniform CAs. We tested the following uniform CA rules – without time spacing: 30, 90, 105, 150, 165. Taking into account all the Diehard tests, including some that are not are shown in Table 2 (the multiple p-value tests), we can conclude the following: rule 105 is the best RNG (among the uniform rules), followed by rules 165, 90, and 150, with rule 30 coming in last. All these CAs failed the bitstream and OPSO tests. With respect to the OQSO test, rule 30 had always failed, while the other rules sometimes produced good (passing) strings. A periodicity of the p-values was detected in all cases. The DNA test differentiated between rules 105, 165, and 150 – which had generally passed it, and rules 30 and 90 – which had mostly failed. We conclude that on the whole, uniform CAs without time spacing comprise fairly good generators, but they do not compare well with standard classical ones (see previous paragraph).

Non-uniform CAs. Three non-uniform CAs were tested (Table 3): Hortensius et al., Sipper and Tomassini, and Tomassini et al. (see Section 3). Our results show that, on the whole, non-uniform CAs (without time spacing) are better RNGs than uniform ones, although they are still somewhat inferior to classical generators; this confirms and extends the findings of [2,3,12]. We observed no discernible differences

Table 2
Diehard p-value test results for uniform CAs without time spacing

Test name	30	90	105	150	165
Birthday spacing	1.000	1.000	1.000	1.000	1.000
Overlapping permutation1	1.000	1.000	1.000	1.000	1.000
Overlapping permutation 2	1.000	1.000	1.000	1.000	1.000
Binary rank 31*31	1.000	1.000	1.000	1.000	1.000
Binary rank 32*32	1.000	1.000	1.000	1.000	1.000
Binary rank 6*8	1.000	1.000	1.000	1.000	1.000
Count the ones	1.000	1.000	1.000	1.000	1.000
Parking lot	1.000	0.381	1.000	1.000	1.000
Minimum distance	1.000	1.000	1.000	1.000	1.000
3D sphere	1.000	0.631	0.395	0.034	0.109
Squeeze	1.000	1.000	0.531	0.278	0.999
Overlapping sum	0.511	0.181	0.396	0.597	0.337
Run up 1	0.880	0.078	0.995	1.000	0.517
Run up 2	0.121	0.940	0.875	0.988	0.274
Run down 1	0.976	0.602	0.659	1.000	0.638
Run down 2	0.703	0.730	1.000	0.988	0.568
Craps number of throws	1.000	0.021	0.827	1.000	0.112
Craps number of wins	1.000	0.328	0.952	0.857	0.836

Table 3
Diehard p-value test results for non-uniform CAs without time spacing

Test name	Hortensius et al.	Sipper and Tomassini	Tomassini et al.
Birthday spacing	1.000	1.000	1.000
Overlapping permutation 1	0.030	0.953	0.990
Overlapping permutation 2	0.028	0.956	0.546
Binary rank 31*31	1.000	1.000	1.000
Binary rank 32*32	1.000	1.000	1.000
Binary rank 6*8	0.425	0.168	0.763
Count the ones	1.000	1.000	1.000
Parking lot	0.900	0.089	0.478
Minimum distance	1.000	1.000	1.000
3D sphere	0.910	0.058	0.948
Squeeze	0.920	0.633	0.781
Overlapping sum	0.554	0.478	0.011
Run up 1	0.722	0.327	0.399
Run up 2	0.646	0.210	0.604
Run down 1	0.485	0.782	0.994
Run down 2	0.211	0.193	0.646
Craps number of throws	0.303	0.144	0.590
Craps number of wins	0.832	0.359	0.255

between the three non-uniform CAs tested, with all of them failing to pass the same Diehard tests. We also detected periodicity in the p-values, which means that their generation of random sequences leaves something to be desired. Note, though, that they did pass a number of hard tests, and can therefore serve as RNGs in many applications. Their advantage is that they produce more random numbers per time unit, since no time spacing is used.

Time spacing. Because of the observed p-value periodicities in the above uniform and non-uniform CAs, we applied a time-spacing parameter of 5, obtaining very good results. However, such time spacing implies that only one bit in six is retained, thus markedly decreasing the rate at which random numbers are produced. As an acceptable compromise between bit rate and quality, we opted for a time spacing of 2. Our results show highly improved performance both for the uniform (Table 4) and the non-uniform (Table 5) cases, with but a small degradation with respect to a time spacing of 5. Indeed, it can be seen that these CA generators are as good as the standard ones (Table 1). The five uniform CAs all exhibit roughly the same performance level. Among the three non-uniform CAs, Sipper and Tomassini and Tomassini et al. are roughly equivalent (though the latter is slightly better, due to some multiple p-value tests not shown here), while Hortensius is slightly worse in this respect. In conclusion, if one seeks very high-quality random number sequences, then time spacing seems to be crucial. With time spacing, there seems to be little difference between uniform and non-uniform CAs.

Cross-correlation.

- *Cross-correlation between different CAs.* We tested two pairs of CAs: the cross-correlation between the rule-105 CA and the Tomassini et al. CA, and the cross-correlation between the Sipper and Tomassini CA and the Tomassini et al. CA. In both cases, no correlations were found between the sequences generated by each CA of the pair (for the entire 80MB file).
- *Cross-correlation between sequences from the same CA with different initial conditions.* In this case we use the same CA, but test for cross-correlation between two sequences created from two different initial state configurations. We tested uniform rule 105, Sipper and Tomassini, and Tomassini et al.. We detected cross-correlations between the sequences generated by the first two CAs, whereas Tomassini et al. showed no such correlations. These two cross-correlation tests are important where parallel machines are concerned. They demonstrate that one can use some of our CAs to generate

Table 4
Diehard p-value test results for uniform CAs with a time spacing of two

Test name	30	90	105	150	165
Birthday spacing	0.447	0.821	0.159	0.028	0.605
Overlapping permutation 1	0.818	0.176	0.322	0.523	0.143
Overlapping permutation 2	0.363	0.828	0.227	0.015	0.043
Binary rank 31*31	0.572	0.321	0.480	0.329	0.620
Binary rank 32*32	0.610	0.698	0.354	0.500	0.528
Binary rank 6*8	0.456	0.939	0.431	0.841	0.061
Count the ones	1.000	0.787	0.723	0.395	0.464
Parking lot	0.455	0.164	0.210	0.917	0.577
Minimum distance	1.000	0.100	1.000	0.995	0.988
3D sphere	0.054	0.891	0.749	0.870	0.242
Squeeze	0.620	0.846	0.502	0.478	0.285
Overlapping sum	0.165	0.519	0.261	0.132	0.300
Run up 1	0.775	0.755	0.131	0.385	0.904
Run up 2	0.424	0.711	0.176	0.000	0.407
Run down 1	0.361	0.880	0.749	0.558	0.861
Run down 2	0.414	0.842	0.807	0.295	0.719
Craps number of throws	0.167	0.715	0.979	0.484	0.551
Craps number of wins	0.786	0.192	0.312	0.225	0.484

Table 5
Diehard p-value test results for non-uniform CAs with a time spacing of two

Test name	Hortensius et al.	Sipper and Tomassini	Tomassini et al.
Birthday spacing	0.997	0.287	0.647
Overlapping permutation 1	0.969	0.053	0.094
Overlapping permutation 2	0.971	0.944	0.373
Binary rank 31*31	0.649	0.353	0.958
Binary rank 32*32	0.481	0.368	0.928
Binary rank 6*8	0.213	0.768	0.721
Count the one's	1.000	0.232	0.639
Parking lot	0.307	0.055	0.991
Minimum distance	1.000	0.974	1.000
3D sphere	0.121	0.391	0.977
Squeeze	0.281	0.256	0.557
Overlapping sum	0.677	0.304	0.231
Run up 1	0.615	0.856	0.186
Run up 2	0.505	0.335	0.374
Run down 1	0.149	0.003	0.777
Run down 2	0.760	0.539	0.557
Craps number of throws	0.623	0.429	0.698
Craps number of wins	0.681	0.081	0.452

uncorrelated sequences on different machines, running in parallel (e.g., parallel Monte-Carlo simulations).

- *Cross-correlation between interleaved sequences from the same CA with a given time spacing parameter.* Here, the CA is used with a time spacing of 2, which essentially creates three random number sequences, starting at time steps 0, 1, and 2, respectively. Normally, with time spacing, only the first one is retained, and our interest here was to check whether two (or more) of these could be used in parallel. This is important, e.g., for VLSI built-in self-test (BIST) applications since one can use the same CA to create two or more random

Table 6
Number of tests each RNG passed

Random number generator	Number of tests passed
Tomassini et al. with time spacing	18
Sipper and Tomassini with time spacing	18
Rule 165 with time spacing	18
Rule 90 with time spacing	18
RAN3	18
Rule 105 with time spacing	17
Rule 150 with time spacing	17
Hortensius with time spacing	16
Rule 30 with time spacing	16
GGL	16
Tomassini et al.	13
Hortensius	13
Sipper and Tomassini	13
Rule 90	9
Rule 165	9
Rule 105	8
Rule 150	6
Rule 30	5

sequences, which can be used by different parts of the tested circuit.

Before studying sequence cross-correlations, we submitted each of the three sequences separately to the Diehard test suite in order to assess their randomness. The results were good and comparable to the single-sequence ones described above. Testing CAs 105, Tomassini et al., and Sipper and Tomassini we found that the first two exhibited good performance, whereas the latter one turned out to be worse.

7. Concluding remarks

Table 6 summarizes our findings, ranking all tested RNGs according to the quality of the random numbers they produce. In general, non-uniform CAs are better than uniform CAs without time spacing. Since this is the fastest method of producing random numbers, such CAs are the RNG of choice. Though they are somewhat inferior to linear congruential and lagged-Fibonacci ones, the quality of the random number sequences produced is nonetheless quite high, and is sufficient for many applications. Thus, non-uniform CAs with no time spacing provide for good, fast RNGs.

Remember that the Tomassini et al. CA was created by assigning to each cell one of the four "good" rules (discovered by evolution) at random – 90, 105, 150, or 165. This suggests that good RNGs of any size might be obtained using this simple procedure.

Time spacing improves the quality of the random number sequences, both for uniform and non-uniform CAs. Our findings show that a time spacing of 2 is sufficient in order to match the quality of good classical RNGs (linear congruential and lagged Fibonacci). Thus, one can trade off random bit generation rate with sequence quality, reducing the former in order to improve the latter.

The cross-correlation tests are important where parallel sequence generation is concerned. Our results show that one can use some of our CAs to generate uncorrelated sequences on machines running in parallel.

In conclusion, our extensive suite of tests demonstrates that CAs can be used to rapidly produce high-quality random number sequences. Such CAs can be efficiently implemented in hardware, can be used in such applications as VLSI built-in self-test, and can be applied in the field of parallel computation.

Acknowledgements

The authors would like to thank W. Peterson for useful discussions.

References

[1] D.R. Chowdhury, I.S. Gupta, P.P. Chaudhuri, A low-cost high-capacity associative memory design using cellular automata, IEEE Trans. Comput. 44(10) (1995) 1260–1264.
[2] P.D. Hortensius, R.D. McLeod, H.C. Card, Parallel random number generation for VLSI systems using cellular automata, IEEE Trans. Comput. 38(10) (1989) 1466–1473.
[3] P.D. Hortensius, R.D. McLeod, W. Pries, D.M. Miller, H.C. Card, Cellular automata-based pseudo-random number generators for built-in self-test, IEEE Trans. Computer-Aided Design 8(8) (1989) 842–859.
[4] D.E. Knuth, The Art of Computer Programming: Vol. 2, Seminumerical Algorithms, Addison-Wesley, Reading, MA, 3rd ed., 1998.
[5] J.R. Koza, Genetic Programming, The MIT Press, Cambridge, Massachusetts, 1992.
[6] D. Mange, M. Tomassini, editors. Bio-inspired Computing Machines. Presses Polytechniques et Universitaires Romandes, Lausanne, 1998.

[7] G. Marsaglia, A current view of random number generators, in: L. Billard (Ed.), Computer Sciences and Statistics, pp. 3–10. Elsevier, Amsterdam, 1985.
[8] G. Marsaglia, Diehard. http://stat.fsu.edu/~geo/diehard.html, 1998.
[9] Z. Michalewicz, Genetic Algorithms + Data Structures = Evolution Programs. Springer, Heidelberg, 3rd ed., 1996.
[10] S.K. Park, K.W. Miller, Random number generators: Good ones are hard to find, Communications of the ACM 31(10) (1988) 1192–1201.
[11] M. Sipper, Evolution of Parallel Cellular Machines: The Cellular Programming Approach, Springer, Heidelberg, 1997.
[12] M. Sipper, M. Tomassini, Generating parallel random number generators by cellular programming, Intl. J. Modern Phys. C 7(2) (1996) 181–190.
[13] I. Vattulainen, New test of random numbers for simulations in physical systems. Technical Report HU-TFT-IR-94-4, Research institute for theoretical physics, University of Helsinki, 1994.
[14] S. Wolfram, Random sequence generation by cellular automata, Adv. Appl. Mathematics 7 (1986) 123–169.

Marco Tomassini is professor of Computer Science at the University of Lausanne, Switzerland. His research interests include bio-inspired techniques, such as evolutionary algorithms and artificial neural networks, as well as heuristics, machine learning, and knowledge discovery. He is also involved in the study of cellular automata and of complex systems of agents in economics and in information and communication webs. He has authored and coauthored several scientific papers and books in these fields.

Moshe Sipper is a senior researcher in the Logic Systems Laboratory at the Swiss Federal Institute of Technology, Lausanne, Switzerland. He received a B.A. in Computer Science from the Technion-Israel Institute of Technology, and M.Sc. and a Ph.D. from Tel Aviv University. His chief interests involve the application of biological principles to artificial systems, including, cellular automata (with an emphasis on evolving cellular machines), bio-inspired systems, evolving hardware, complex adaptive systems, artificial life, and neural networks. Dr. Sipper has authored and coauthored several scientific papers in these areas, as well as the book *Evolution of Parallel Cellular Machines: The Cellular Programming Approach* (Springer, Heidelberg, 1997).

Mosè Zolla is a graduate student at the Computer Science Institute of the University of Lausanne. After a diploma in chemistry and computer science, he is studying the statistical properties of cellular automata-based pseudo-random numbers. His main interests include statistical methods such as bootstrap and computational statistics.

Mathieu Perrenoud is a graduate student at the Computer Science Institute of the University of Lausanne. He is studying artificial evolution and cellular automata and is interested in artificial intelligence and the world wide web. He also enjoys playing bridge and science fiction movies.