On the Generation of High Quality Random Number by Two-Dimensional Cellular Automata Using Advanced Cell-DEVS Models with CD++

Abdulelah Al-Sanad
Student ID: 3521674
aalsa079@site.uottawa.ca
SITE, University of Ottawa

Term Projects for SYSC 5104
Methodologies for Discrete-Event Modelling and Simulation
Fall 2005

December 12, 2005

Contents

1 Introduction

Random numbers are used in a wide variety of applications that could could be in scientific, mathematical, engineering, or industrial applications [1]. For example, they are used in cryptography to provide network security, and in many optimization algorithms such us simulated annealing. The performance of these applications depends highly on the quality of the random numbers being generated for them. Therefore, random number generators plays an essential role in many practical systems in our life. However, finding a good random number generator is not an easy task as will be explained in following section.

Testing tools are used to evaluate the randomness of the numbers generated using statistical properties and measurement [2]. Good random number generators has to produce numbers that are uniformly distributed and at the same time there should not be any correlation between these numbers. Also, the numbers should have long periods such that they do not produces themselves again [3].

There are several approaches used to generate random number. For example, linear feedback shift register is one of the most common used methods as a random number generator [4]. Other approaches are based on mathematical equations to generate random number. Cellular automata models can also be used to generate random numbers. It have been proved that the randomness of the numbers generated by cellular automata have better quality compared with other techniques. It is assumed that the readers are aware of cellular automata systems.

1.1 Using cellular automata models to generate random numbers

There are many advantages that makes cellular automata better than other methods [5]. The followings are some advantages to use cellular automata for generating random numbers:

- **Simplicity:** cellular automata is a simple model that just consist of cells with state variables. A cell changes its state based on the rules applied to it and its neighbors.
- **Regularity:** cells in cellular automata are arranged in a regular manner that could have one or more dimensions.
- **Locally interconnected:** a cell in cellular automata is connected locally with its neighbors.

- **Easier to implement in hardware:** because of their simplicity, regularity, and interconnection, cellular automata are easy to implement in hardware and in other recent technology such as FPGA.
- **Produce random number rapidly:** cellular automata can also generate random number in very short time compared with other methods.

1.2 Contribution of the project

In this project several models are implemented and simulated with different properties to generate random numbers. The results of these models are analyzed to test the behaviour of the randomness of the number generated. New techniques are proposed to enhance the quality of the random number generated such as changing the neighborhood size for two-dimensional cell model, the way the number extracted from the model and applying different delays for the rules. The results of the simulation shows out these enhancements proposed to the models, help in producing better randomness in the number generated.

2 Background

Generating random number using one dimensional cellular automata models have been studied extensively in the past. Recent researches nowadays focus in two dimensional cellular models in order to provide higher quality random number generator at the expense of adding a little bid of complexity [6].

2.1 Quality of the random number generated by cell automata models

There are four properties of cellular automata models that can affect the quality of random number generators. These properties are:

- **Uniform vs. non-uniform models:** When the same rules are applied to all the cells, then this is called uniform model. However, when different rules are applied to different cell randomly, then this is called non-uniform model. Both models are simulated in this project. Non-uniform models produce better randomness to the number generated than uniform models.
- **Boundary condition:** The boundary of the models can affect the randomness of the number generated by that models. Two condition are simulated: wrapped and non-wrapped models.

Table 1: Rules are specified in the form of rule table

Rule Name	Possible Input Configuration							
	111	110	101	100	011	010	001	000
90	0	1	0	1	1	0	1	0
165	1	0	1	0	0	1	0	1
150	1	0	0	1	0	1	1	0
105	0	1	1	0	1	0	0	1

- **Length of the cellular model:** The size of the model can also affect the quality of the randomness of the numbers. However, by increasing the length of the models, the time it takes to generate the random number will also be longer. Therefore, a trad off should be made to choose the optimal size.
- **Initial configuration:** The initial configuration of the cells in the model is another factor that can affect the randomness of the numbers.

2.2 Rules for random number generator

The rules or the transition function are specified in the form of rule tables. Then, the table has an entry for every possible configuration of the states of the neighbourhood. Therefore, for a neighborhood of size 5, there will be $2^5 = 32$ possible configurations of the states of these cells which will give 2^{32} possible rules might be applied. Therefore, for cell cells of 8×8, the search space for the best rules will be $(2^{32})^{64}$ which is extremely high. As a result, in order to find the rules which generate a good random number, an approximation algorithm or a heuristic is required to do the searching.

A rule numbering scheme is used to describe the rules by encoding the rule table using binary number. For example, for one-dimensional cellular model with a neighbourhood size of 3, there are 2^3 possible configurations of the states and 2^8 possible rules. Then, these 256 rules are numbered from 1 to 256 by encoding the output of these rules as shown in Table 1 for some of the rules.

3 Models defined

The states of the cells are of type boolean, therefore, $S_i \in \{0, 1\}$. Two-dimension 8×8 cellular model with a neighbourhood size of 5 cells is simulated.

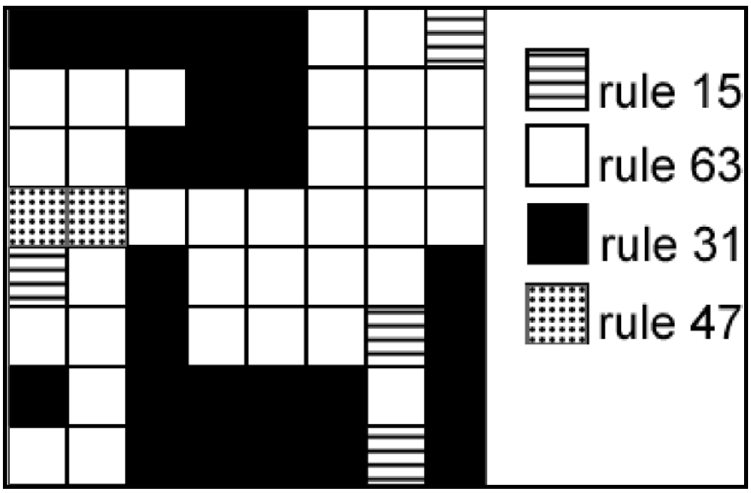

Figure 1: The rules map showing the best rules that generate high-quality random number [7].

Rules that give rise to high-quality sequence of random number are derived from [7]. The rules map are shown in Figure 1

Reference [7] claims that rules : 15, 63, 31, and 47 are the best to use for generating random number due to the mathematical characteristic of the pattern that the generate. These rules are applied randomly to the cells for more randomness.

These rules can be written in a form of boolean equation for more efficient execution of the rules. Let $s_{i,j}(t)$ be the state of the currant cell at row i and column j at time t, then the next state for this cell at the next time step for rule 15 is defined as:

$$s_{i,j}(t+1) = s_{i-1,j}(t) \oplus s_{i,j-1}(t) \oplus s_{i+1,j}(t) \oplus s_{i,j+1}(t) \tag{1}$$

Also, rule 31 can be applied using the equation:

$$s_{i,j}(t+1) = s_{i-1,j}(t) \oplus s_{i,j-1}(t) \oplus s_{i+1,j}(t) \oplus s_{i,j+1}(t) \oplus s_{i,j}(t) \tag{2}$$

Similarly, the following equation represents rule 47:

$$s_{i,j}(t+1) = 1 \oplus s_{i-1,j}(t) \oplus s_{i,j-1}(t) \oplus s_{i+1,j}(t) \oplus s_{i,j+1}(t) \tag{3}$$

Moreover, rule 63 is defined by:

$$s_{i,j}(t+1) = 1 \oplus s_{i-1,j}(t) \oplus s_{i,j-1}(t) \oplus s_{i+1,j}(t) \oplus s_{i,j+1}(t) \oplus s_{i,j}(t) \tag{4}$$

3.1 Formal specifications for the Cell-DEVS models

The formal specification for the model is as follows:

$M =< I, X, Y, X_{list}, Y_{list}, \eta, N, m, n, C, B, Z, select >$
$X_{list} = \emptyset;$
$Y_{list} = \emptyset;$
$\eta = 5;$
$I =< P^x, P^y >, with P^x = \emptyset; P^y = \emptyset;$
$N = (-1, 0), (0, -1), (0, 0), (0, 1), (1, 0);$
$X = 0, 1;$
$Y = 0, 1;$
$m = 8;$
$n = 8;$
$B = \emptyset;$
$D = 1 - 100;$
$C = C_{Aij}/i \in [1, 8], j \in [1, 8];$
$Z :$
$P_{ij}Y_1 \longrightarrow P_{i,j-1}X_1 \quad P_{i,j+1}Y_1 \longrightarrow P_{ij}X_1$
$P_{ij}Y_2 \longrightarrow P_{i+1,j}X_2 \quad P_{i-1,j}Y_2 \longrightarrow P_{ij}X_2$
$P_{ij}Y_3 \longrightarrow P_{i,j+1}X_3 \quad P_{i,j-1}Y_3 \longrightarrow P_{ij}X_3$
$P_{ij}Y_4 \longrightarrow P_{i-1,j}X_4 \quad P_{i+1,j}Y_4 \longrightarrow P_{ij}X_4$
$P_{ij}Y_5 \longrightarrow P_{ij}X_5 \quad P_{ij}Y_5 \longrightarrow P_{ij}X_5$
$select = (-1, 0), (0, -1), (0, 0), (0, 1), (1, 0);$

3.2 Enhancement techniques proposed for better result

There are many different possible implementations that can affect the quality of randomness in the number generated. In this section, some of these techniques that are proposed, are explained.

3.2.1 Changing the neighbourhood size

In the model defined above, only five neighbor cells are considered. However, for two-dimensional cellular models, there could be 8 neighbor of a cell which will give a neighbourhood of size 9. By increasing the size neighborhood, more cells will be involved in determining the next state which might increase the quality of randomness. However, this will also add more complexity to the model which will contradict with the constraint in random number generator to keep them simple.

3.2.2 Extracting the number

The method used to extract the number from the model can also affect the quality of the number. For example, not all bits can be considered as part of the random number by skipping some cells in the model. This can be done randomly or by predefining the locations of the cells that are involved in the process.

3.2.3 Applying different delays

By changing the delays of the rules to produce the results, the quality of the random number can be more efficient. Most of the models proposed in the literature used fixed time delay for the rules. Therefore, making the delay to be variable is tested in this project to check its effect on the model to generate random numbers. The *randInt* function is used to generate a random number for the delay between 1 and 100 as shown below:

```
rule : 0 {randInt(100)} { ((-1,0) = 1 and (0,-1) = 1 and (1,0) = 1
and (0,1) = 1 and (0,0) = 1) and (if(randInt(3) = 1,1,0) = 1)}
```

3.3 Building the Cell-DEVS model using CD++

A cell-DEVS model is built using CD++ to generate random number. A truth table is constructed for each of the equations used to define the rules. For example, the equation for rule 15 has four operands with XOR operations. Therefore, a truth table of size 2^4 is required to execute that equation. For example the macro below shows the rules used to implement rule 15.

```
#BeginMacro(rng-rule15)
rule : 1 randInt(100) { ((-1,0) = 1 and (0,-1) = 1 and (1,0) = 1
                    and (0,1) = 1) and (if(randInt(3) = 2,1,0) = 1)}
rule : 0 randInt(100) { ((-1,0) = 1 and (0,-1) = 1 and (1,0) = 1
                    and (0,1) = 0) and (if(randInt(3) = 2,1,0) = 1)}
rule : 0 randInt(100) { ((-1,0) = 1 and (0,-1) = 1 and (1,0) = 0
                    and (0,1) = 1) and (if(randInt(3) = 2,1,0) = 1)}
rule : 1 randInt(100) { ((-1,0) = 1 and (0,-1) = 1 and (1,0) = 0
                    and (0,1) = 0) and (if(randInt(3) = 2,1,0) = 1)}
rule : 0 randInt(100) { ((-1,0) = 1 and (0,-1) = 0 and (1,0) = 1
                    and (0,1) = 1) and (if(randInt(3) = 2,1,0) = 1)}
rule : 1 randInt(100) { ((-1,0) = 1 and (0,-1) = 0 and (1,0) = 1
                    and (0,1) = 0) and (if(randInt(3) = 2,1,0) = 1)}
```

```
rule : 1 randInt(100) { ((-1,0) = 1 and (0,-1) = 0 and (1,0) = 0
                    and (0,1) = 1) and (if(randInt(3) = 2,1,0) = 1)}
rule : 0 randInt(100) { ((-1,0) = 1 and (0,-1) = 0 and (1,0) = 0
                    and (0,1) = 0) and (if(randInt(3) = 2,1,0) = 1)}
rule : 0 randInt(100) { ((-1,0) = 0 and (0,-1) = 1 and (1,0) = 1
                    and (0,1) = 1) and (if(randInt(3) = 2,1,0) = 1)}
rule : 1 randInt(100) { ((-1,0) = 0 and (0,-1) = 1 and (1,0) = 1
                    and (0,1) = 0) and (if(randInt(3) = 2,1,0) = 1)}
rule : 1 randInt(100) { ((-1,0) = 0 and (0,-1) = 1 and (1,0) = 0
                    and (0,1) = 1) and (if(randInt(3) = 2,1,0) = 1)}
rule : 0 randInt(100) { ((-1,0) = 0 and (0,-1) = 1 and (1,0) = 0
                    and (0,1) = 0) and (if(randInt(3) = 2,1,0) = 1)}
rule : 1 randInt(100) { ((-1,0) = 0 and (0,-1) = 0 and (1,0) = 1
                    and (0,1) = 1) and (if(randInt(3) = 2,1,0) = 1)}
rule : 0 randInt(100) { ((-1,0) = 0 and (0,-1) = 0 and (1,0) = 1
                    and (0,1) = 0) and (if(randInt(3) = 2,1,0) = 1)}
rule : 0 randInt(100) { ((-1,0) = 0 and (0,-1) = 0 and (1,0) = 0
                    and (0,1) = 1) and (if(randInt(3) = 2,1,0) = 1)}
rule : 1 randInt(100) { ((-1,0) = 0 and (0,-1) = 0 and (1,0) = 0
                    and (0,1) = 0) and (if(randInt(3) = 2,1,0) = 1)}
#EndMacro
```

The random function is used to apply different random delays for the rules. Also, it is used to check to make the model non-uniform by applying the rules randomly to the cells.

4 Simulation Results Analysis

In this section, the outputs of the models are analyzed for different possible implementations of random number generator. After building the model using CD++, the model is tested thoroughly before collecting the result as explained in the following subsection.

4.1 Testing the rules of the model

The rules are tested also separately to make sure that their functionality is correct. For example, the expected output of rule 15 for a certain cell configuration is derived and compared with the actual output of the model. Figure 2 shows one such output of applying rule 15 for all the cells with the

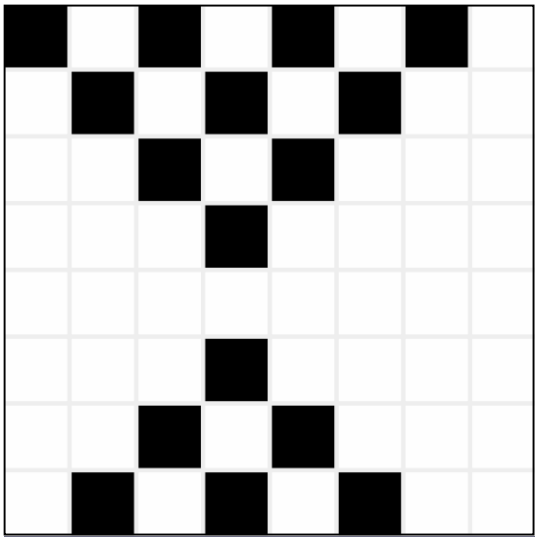

Figure 2:

same delay. For testing, the initial values of the cells are defined such away that the outputs are known.

Figure 3 shows also an example of a result produced by rule 47.

Similarly, the output of rules 31 is shown in Figure 4 and compared with the expected output generated manually.

Moreover, Figure 5 shows the output generated by rule 63 which validate the functionality of this rule.

All the rules are tested with different inputs to validate the functionality of the model. After that, all the rules combined together in one model. However, to apply the rules to different cells, a random function is required to make the model non-uniform.

4.2 Extracting the random number from the model

Once the output of the model is generated, the random number has to be extracted from the model. There are several methods to do this each with its own complexity. The method used here is taken from [7] because of its simplicity and efficiency. In this method, the cell model is traversed four times to get four bits from each cells which might be encoded in hexadecimal number. Thus, for an 8×8 cell model, 64 hexadecimal random digits are produced.

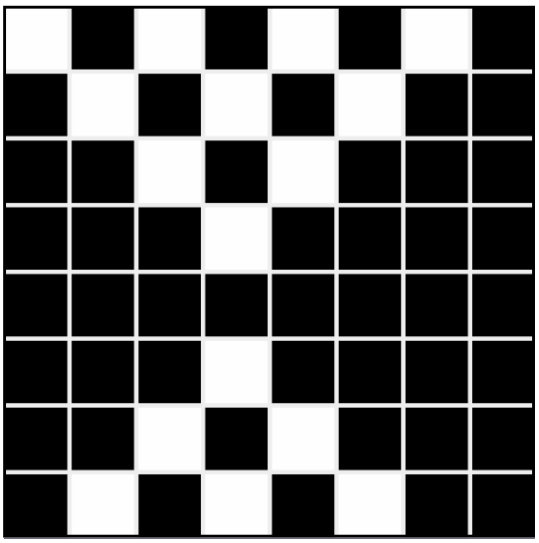

Figure 3: Example of an output for rule 47

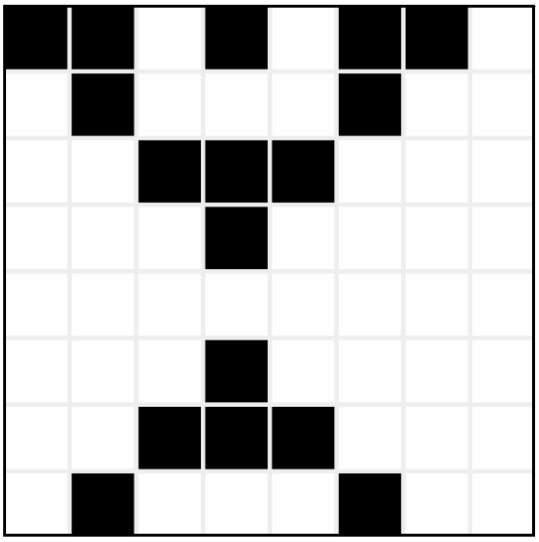

Figure 4: Example of an output for rule 31

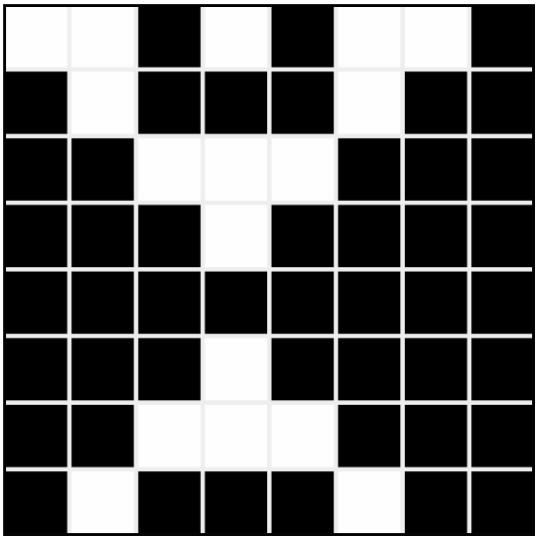

Figure 5: Example of an output for rule 63

4.3 Testing the quality of randomness in the numbers

Several statistical tests are used to measure the quality of a random number. One of the best tests of random sequences is called DIEHARD developed by George Marsaglia [2]. DIEHARD test is used in this project to test the random number generated by the cell model.

DIEHARD test return a p-value on the interval $[0, 1)$ if the input file contains truly independent random bits. This p-value is obtained by $p = F(X)$, where F is the assumed distribution of the sample random variable X.

4.4 Result analysis

The following subsection discusses some of the results generated by the model defined earlier. All the enhancements proposed are tested to check their effects on the randomness of the number.

4.4.1 Uniform vs. non-uniform

Two models different models are simulated and compared: uniform model where the same rule is applied to all of the cells and non-uniform where different rules are applied randomly to the cell. The non-uniform model

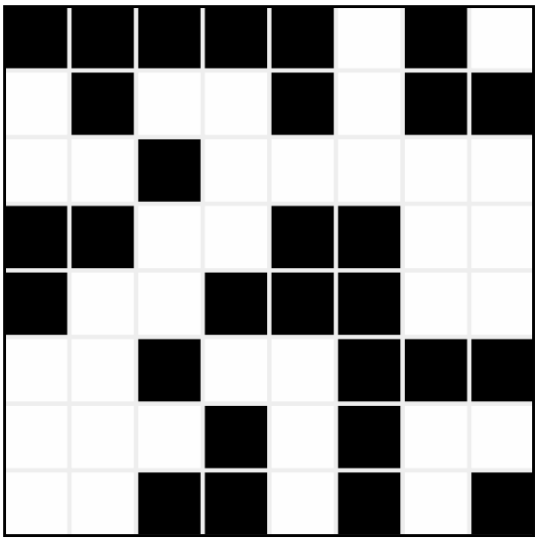

Figure 6: Example of an output for the non-uniform model with the rules 15, 63, 21, and 47.

generate better random number than the uniform model as the DIEHARD test give higher entropy for the non-uniform model.

For example, Figure 6 shows an example of an output generated by the non-uniform model with rules 15, 63, 21, and 47 applied to different cells at random.

4.4.2 Random vs. fixed delay

Another two different models are simulated to check the effect of the delay on the quality of the number. One model uses a random function is used to generate time delay between 1 and 100. The other model has a fix delay for all the rules. The first model with a random delay produces random numbers with higher quality than the other model with fixed delay based on the measurement used by DIEHARD test.

This is can also be proven with output shown in Figure 7 and comparing it with Figure 2. It is very clear from the figures how the randomness of the bits is different by applying different delay for the rules.

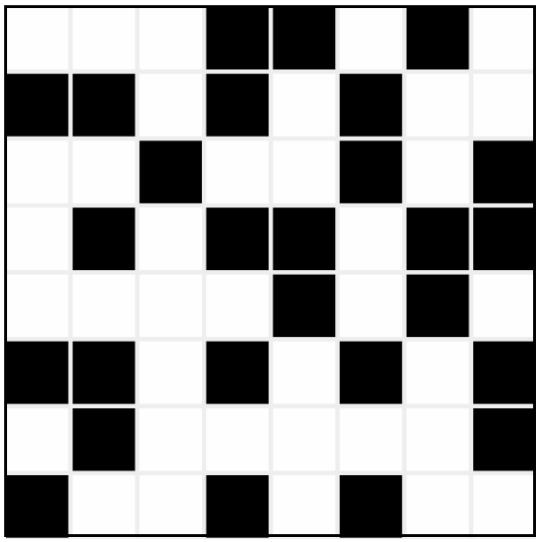

Figure 7: Example of an output for rule 15 with random delay.

4.5 Simulating the model using parallel CD++

The models presented in the previous subsection were implemented using the original version of CD++ that has single state variable. The time it takes to get the result is quite long for the models that uses random rules and random delay. However, since the size of the cell model is small, i.e. 8×8, the execution time is sometime acceptable.

However, a new version of CD++ is now available that supports multiple state variables. It has a tool to execute Parallel Cell-DEVS models in parallel or distributed environments based on Warped and MPI (Message Passing Interface) architectures.

Therefore, the previous models were developed to be compatible with the new version by defining state variable called *value* as the following code shows:

```
statevariables: value
statevalues: 0
initialvariablesvalue: rng.sval
```

The rules are also changed accordingly where the state variable is referenced as shown below:

```
rule : { $value } { $value := 1; } {randInt(100)} { ((-1,0) = 1 and
```

```
(0,-1) = 1 and (1,0) = 0 and (0,1) = 0 and $value = 0) and
(if(randInt(3) = 1,1,0) = 1)}
```

By simulating the new version of the model, the simulation time is reduced compared with the previous model. The speed of simulation can be noticed clearly with the non-uniform model that has random rules and random delay as these type of model took quite long time to execute them in the previous implementing.

5 Conclusions

There are many application that used random numbers and they critically depends on the quality of these random number generated for them. However, finding a good random number generator is a hard problem. It have been shown that two-dimensional cellular automata models can rapidly generate high quality random numbers. There are many different possible implementations that can affect the quality in randomness of these number.

In this project, cell models are simulated to test the quality of the random numbers they generate by applying different techniques to the models. The results of the simulation shows out these enhancements proposed to the models, help in producing better randomness in the number generated.

In conclusion, a high quality random number can be generated using small size two-dimensional cellular model with non-uniform ruling and different delays for the rules.

References

[1] S. Nandi, B. K. Kar, and P. P. Chaudhuri, "Theory and applications of cellular automata in cryptography," *IEEE Transactions on Computers*, vol. 43, pp. 1346 – 1357, Dec. 1994.

[2] G. Marsaglia, "Diehard test," http://stat.fsu.edu/pub/diehard/, 2003.

[3] P. BARDELL, "Analysis of cellular automata used as pseudorandom pattern generators," in *Proceedings of the IEEE 21st International Test Conference*, Sept. 1900, pp. 762–768.

[4] S. Guan and S. Zhang, "An evolutionary approach to the design of controllable cellular automata structure for random number generation," *IEEE Transactions on Evolutionary Computation*, vol. 7, pp. 23–36, Feb. 2003.

[5] M. Sipper, M. Tomasso, M. Zolla, and M. Perrenoud, "Generating high-quality random numbers in parallel by cellular automata," *Future Generation Computer Systems*, vol. 16, pp. 291–305, Dec. 1999.

[6] B. Shackleford, M. Tanaka, R. Carter, and G. Snider, "High-performance cellular automata random number generators for embedded probabilistic computing systems," in *Proc. 4th NASA/DOD Conference on Evolvable Hardware*, July 2002, pp. 191–200.

[7] M. Tomassini, M. Sipper, and M. Perrenoud, "On the generation of high-quality random numbers by two-dimensional cellular automata," *IEEE Transactions on Computers*, vol. 49, pp. 1146–1151, Oct. 2000.

[8] S. U. Guan and S. K. Tan, "Pseudorandom number generation with self programmable cellular automata," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, pp. 1095–1101, July 2004.

[9] M. Matsumoto, "Simple cellular automata as pseudorandom m-sequence generators for built-in self-test," *ACM Transactions on Modeling and Computer Simulation*, vol. 8, pp. 31 – 42, Jan. 1998.