The game of Scintillae: From cellular automata to computing and cryptography systems

GABRIELE DI STEFANO1, ALFREDO NAVARRA2

1 Dipartimento di Ingegneria e Scienze dell’Informazione e Matematica, Università degli Studi dell’Aquila, Via Gronchi 18 I-67100 L’Aquila, Italy. gabriele.distefano@univaq.it
2 Dipartimento di Matematica e Informatica, Università degli Studi di Perugia, Via Vanvitelli 1 I-06123 Perugia, Italy. alfredo.navarra@unipg.it (Contact author)

The paper deals with a very simple game called Scintillae. Like in a domino game, Scintillae provides the player with limited basic pieces that can be placed over a chessboard-like area. After the placement, the game starts in a sort of runtime mode, and the player enjoys his creation. The evolution of the system is based on few basic rules.

Despite its simplicity, Scintillae turns out to provide the player with a powerful mean able to achieve high computational power, storage capabilities and many other peculiarities based on the ability of the player to suitably dispose the pieces.

We show some of the potentials of this simple game by providing basic configurations that can be used as “sub-programs” for composing bigger systems. Moreover, the interest in Scintillae also resides in its potentials for educational purposes, as many basic concepts related to the computer science architecture can be approached with fun by means of this game.

Key words: Cellular automata, computational model, computing systems, visual programming, cryptography, boolean circuit, logic gate

* Preliminary results about this work have been presented in [10]. This work has been partially supported by the Research Grant 2010N5K7EB ‘PRIN 2010’ ARS TechnoMedia (Algoritmica per le Rete Sociali Tecno-mediate) from the Italian Ministry of University and Research.
1 INTRODUCTION

Scintillae is a game designed for fun to simulate a sort of domino effect on a PC.* Although it hides some peculiarities that cannot be realized by standard domino games with falling pieces, it is very interesting how such kind of simulator may easily realize a computing system in a sort of visual programming environment [6]. Its strength is witnessed by the high computational power that indeed can be obtained by means of the few pieces provided by the game for composing desired configurations. Instead of domino pieces falling in a sequence determined by their proximity, Scintillae provides the user with sparks that seem to move according to designed paths obtained by disposing arrows on a chessboard-like area (see Figure 1). The rules that establish the evolving of the system are very simple.

* An executable graphic version of the program along with some explanatory examples can be found in [1, 2]. The latest open-source release can be found in [3].
Rules of Scintillae  Given an unbounded area divided into squares, two basic pieces can be placed, at most one per square, to compose a configuration: Sparks (“Scintillae” in Latin), and four types of Arrows ($\rightarrow$, $\uparrow$, $\downarrow$, $\leftarrow$), see Figure 2. Each arrow has four neighboring squares. The square pointed by the arrow is called the output square while the other three squares are called input squares. The system is synchronous and pieces interact at each time $t$ according to the next simple operations applied sequentially, that move the system’s state to time $t + 1$:

1. an arrow becomes loaded if among its input squares there is exactly one spark;

2. each spark disappears and leaves empty the corresponding square;

3. for each loaded arrow, a spark is placed in the output square, if empty;

4. each loaded arrow becomes unloaded.

Figure 2 also shows state transitions on some basic configurations. By the rules above, it follows that arrows never change from their original placement. That’s all! The rules are specified, now only the imagination of the user can lead to surprising results. In the implemented programs, once the pieces have been placed, a running button can be pressed to enjoy the evolution.

The duplication of a spark comes from the rules specified above. In fact, if a spark is neighboring the tail of many arrows (at most four), at the successive clock’s tick, one spark per arrow may appear in the output squares pointed by...
FIGURE 3
On the left, the well-known Gosper Glider Gun construction from the Game of Life tool. On the right, the twelve cells composing the von Neumann neighborhood of radius 2 on which depends the status of the central cell in Scintillae.

the arrows according to the described rules. This duplication behavior is very important for obtaining surprising effects. Clearly, Scintillae belongs to the family of Cellular Automata [7, 9, 13, 14, 16, 19]. On the left side of Figure 3, for instance, it is shown a snapshot of a well-known configuration from the Game of Life [4, 11], that has been recently commemorated by a survey book on cellular automata [5]. Similarly to such models, in Scintillae the user is provided with an area divided into cells. Each cell may change its status according to its own one, and to the status of a limited number of neighbors. The evolution is synchronous for all the cells. Hence, Scintillae preserves the fundamental peculiarities of such kind of computational means, that are: parallelism, locality, and homogeneity. In fact, all the cells are updated synchronously in parallel, by applying the same common rule based on local peculiarities. In particular, as previously defined, the status of a cell in Scintillae at time $t$ may depend on the so-called von Neumann neighborhood of radius 2, that is the one shown in the right side of Figure 3. If a cell contains an arrow, it will never change. If a cell is empty or contains a spark, then, at time $t + 1$, it will contain a spark if and only if the following condition holds. At least one of the neighboring cells - denoted in the Figure by 1, 2, 3, and 4 - contains an arrow pointing to the considered central cell such that there is exactly one spark among the input squares of such an arrow.

Likewise domino games but differently from most cellular automata, in Scintillae there are predefined paths provided by the initial disposal of the
FIGURE 4
Three possible cycles with periods 8, 4, and 2, respectively.

arrows that determine the evolution of the sparks. It reminds the WireWorld tool [9], but it is even simpler as less rules are specified. Moreover, the composed configurations appear very neat, clear and intuitive, hence more appropriate for educational purposes. However, “unexpected” but useful behaviors may occur for particular configurations.

2 BASIC CONFIGURATIONS

One main peculiarity of Scintillae is the opportunity to obtain infinite evolutions. In fact, if the user defines a cycling sequence of arrows, and place a spark nearby one of them, the spark will be moved along the cycles infinitely often, until the user stops the run-time mode. In Figure 4, the first three configurations represent possible cycles. The fourth and the fifth configurations shown how cycles can be used as infinite generators of sparks. In particular, the size of the used cycle determines the frequency of sparks in output. The frequency can be modified also by inserting more sparks in the cycle at the beginning during the designing mode.

Interesting basic configurations concern the appropriate placement of the pieces in order to realize desired connections. In fact, a spark can be moved where desired by defining a route provided by a sequence of arrows so that between the head and the tail of two consecutive arrows there is just one square. Then, it might be required that two sequences of arrows (lines) cross each other in order to move different sparks towards desired destinations. When defining two crossing lines, it must be defined a way to realize a “bridge”, so that the two lines do not interfere each other. That is, sparks on one line do not move to the crossing line, nor interfere with sparks moving on the other line. Surprisingly, Scintillae provides an easy way to realize bridges. As shown in Figure 5, we propose two different ways for realizing a bridge between
two lines A and B. It is interesting to observe during the run-time mode how sparks posed at the tails of the first arrows of lines A and B follow the appropriate route without interfering. The first crossing scheme of Figure 5 has been exploited in the circuit of Figure 11.

A useful configuration that recalls a candle or an on/off switch is shown in Figure 6. If a spark is placed nearby the leftmost arrow, the system moves to the rightmost configuration in few clock’s ticks. The configuration remains the same until another spark enters the system. If this happens, again the first configuration occurs (switching off the candle). The rationale behind such a structure is provided by the two sub-structure depicted in the figure. First, an input spark is duplicated by means of the sub-structure $X$. Then, the two consecutive sparks arrive to the cycle in the sub-structure $Y$. If the candle is off (i.e., the cycle is empty), then the effect of the two sparks is that of filling the cycle, and hence the candle turns on. If the candle was on, then the two consecutive sparks remove the ones contained in the cycle since the entering arrow of the cycle will be neighbor of two sparks for two steps. Another way for realizing the candle is shown in Figure 7. However, this configuration reveals a weird side effect. In fact, if a spark is placed near by the leftmost arrow, the system moves to an instable situation where the candle is switched on, but the configuration alternates between the second and the third ones shown in the figure. When another spark enters the system, the candle is switched off, but according to the time this new spark arrives, the configuration might become either the original one or the fourth one.

Clearly, the configurations shown so far represent only few samples with respect to the potentials of Scintillae. The ability of the user to find new ways
of composing the available pieces might realize surprising configurations.

3 FROM THE GAME TO COMPUTING

In this section, we exploit some of the potentials of Scintillae in order to realize computing systems. An interesting peculiarity of Scintillae is its nice attitude to provide a way for realizing combinatorial circuits. We now show how to implement the basic logic gates like XOR, OR, NOT and AND.

The most natural logic gate obtainable by composing few lines of arrows is the XOR. As shown in Figure 8, it is enough to put a line of arrows between the two lines carrying the two inputs of the gate. In fact, the first arrow of the output line will move a spark towards the output only if one single spark arrives from the input lines. When both the input lines carry on a spark, then the first arrow of the output line will be neighboring two sparks that disappear at the next clock’s tick, as shown in the fifth configuration of Figure 2.
FIGURE 8
XOR, OR, and NOT realized on Scintillae.

Remark: Actually, a single arrow in Scintillae represents a XOR gate of three inputs, the neighboring squares not pointed by the arrow. It follows that any other logic gate obtainable by composing Scintillae’s pieces is the result of composing XOR gates. Although the XOR is not a universal gate like the NOR gate (see [15]), i.e., not all the other logic gates can be obtained from the XOR, we show how this problem can be overcome in Scintillae. Even tough this seems a contradiction, it will be better clarified later on.

Realizing the OR gate is also quite easy since it requires a little modification with respect to the XOR, as shown in Figure 8. Now, if both the input lines carry on a spark, they will be both moved to the tail of the output line, where it appears like only one spark as output. In order to realize the NOT, a bit more understanding is required. In fact, we need to have a spark in the output line when there is nothing in input. In order to realize such a configuration, we make use of an infinite generator of sparks, as shown in the fifth configuration of Figure 4. As shown in Figure 8, when the input line carries on a spark, the middle arrow of that line will be neighboring with two sparks, as in the sixth configuration of Figure 2, and then there won’t be a spark at its head in the next clock’s tick.

Remark: In the construction of the NOT gate resides the trick to obtain any other logic gate by means of XOR gates. In fact, once we have both the OR and the NOT gates we can obtain any other gate. The key-point is that we are using also sparks to realize the NOT gate. In particular, we are able to generate a sequence of infinite sparks representing a line set to 1, and this is not possible when considering only XOR gates.

Concerning the AND gate, by simply applying De Morgan’s rule [15], it can be realized as NOT(NOT(A) OR NOT(B)). This is shown in the first configuration of Figure 9. In the same figure, the other configuration still realizes the AND gate but some more attention is required for its understanding. As
part of the game, we tried to obtain the same results by means of optimized configurations in terms of used pieces. For the case of the AND gate, for instance, while the first configuration of Figure 9 is quite straightforward since obtained by applying well-known rules, the other comes from our intuitions. Indeed, the rationale behind it is simply to merge the infinite generators of the first configuration as much as possible. More tricky configurations that realize the AND gate are shown in Figure 10. These required much more confidence with the game.

Another interesting circuit obtained by serializing some candles and appropriately connecting them with NOT gates, realizes a binary counter as shown in Figure 11. The assumption is that 1 corresponds to the candle switched on, 0 otherwise. In the figure, it is shown a snapshot of the system after injecting eleven sparks as input (rightmost arrow) starting from a
configuration representing 0, that is without sparks on the output lines.

In Figure 12, we show a memory component of one byte with recorded value 11011100. The byte is contained in the cycle $M$. In order to read the value contained in the memory, one has to insert a spark in the read line $R$. For a correct reading of the byte, the input spark must arrive at line $R$ synchronously with a spark that in the CLOCK cycle occupies the bottom right-most square as in the figure. Then a copy of the correct sequence describing the byte will flow on the output line $O$. This is realized by temporarily open the “tap” $T3$ between $M$ and line $O$. A tap like $T3$ is composed of a candle and two paths that leads to the same arrow belonging to a path. A closed tap is realized by switching on the candle and hence making two sparks neighboring to the common arrow of a line, i.e., blocking the flow on that line. To open the tap, it is sufficient to switch off the candle, hence removing the block. Note that, on the way from $R$ to $T3$, the line $P2$ is used to duplicate the input spark in such a way that, after eight clock’s ticks, the tap is automatically closed, hence letting flow only eight bits on the output line $O$. The input line $C$ is used to clean the memory. In fact, when a spark enters this way, it is duplicated by means of line $P1$ in such a way that the two sparks reach the tap $T1$ and make it closed for exactly eight clock’s ticks. In doing so, two sparks become neighbors to a common arrow of $M$, hence obtaining the deletion of the contained byte. By using the $C$ line, one obtains also another effect, that is, to open tap $T2$ for exactly eight clock’s ticks. This is, in fact, used for writing in the memory. To write a new byte in the memory, the sequence of sparks describing the new byte must arrive at the input line $W$ concurrently with a spark at line $C$. Hence, $M$ is first cleaned and then

FIGURE 11
A binary 4-bits counter.
refilled with the new sequence. For the synchronization of such operations we were required to carefully consider the length of the involved lines.

4 COMBINATORIAL CIRCUIT CASE STUDY

Based on some previous configurations, we might be able to simulate any combinatorial circuit. In this section, we sketch on the construction of a circuit that counts from 0 to 3, cyclically, and displays the outcome on a standard seven-bars display. Moreover, we make also the display by means of the Scintillae’s pieces. In doing so, we provide the evidence that Scintillae can be used for both computing and wider purposes more related to aesthetical factors. In Figure 13, the mentioned configuration is shown. The snapshot is taken while the counter is displaying the number 2 on the seven-bars display realized by means of arrows. In order to realize the desired counter we need four main sub-circuits: (i) a clock that frequently generates a signal in order to advance the counting; (ii) a counter that translates the received signal from the clock into a binary string representing how many signals have been
received modulo the size of the counter (four in our case); (iii) a seven-bars display that has seven input lines for switching on the correspondent bars; (see Figure 14); (iv) a circuit able to convert the binary string representing a digit among the set \{0, 1, 2, 3\} into the suitable signals that switch on the appropriate bars on the display for a correct visualization. Figure 13 shows the circuit along with the activation functions of the seven-bars display realized.

Circuit (i) is obtained by a cycle of arrows with one spark inside (see bottom left side of Figure 13). In this way, we generate a spark every time the one in the cycle covers all the cycle. Clearly, the more is the length of the cycle, the less is the speed of the counter. The output of such a cycle is connected to a 2-bits counter similar to the one shown in Figure 11, which implements (ii) (see on the left side of Figure 13). The construction realizing (iii) can be easily recognized on the right part of Figure 13, with its seven input lines surrounding the display. This has been realized by means of a suitable disposal of the arrows, some of which are there only for aesthetical reasons. Concerning (iv), the circuit for converting the binary string into switching signals for the seven-bars display is shown in Figure 14, along with the activation
functions related to the bars of the display. Such a circuit is a bit hidden in Figure 13, but following the lines from the counter to the display, one can easily recognize the logic gates used.

5 CRYPTOGRAPHY CASE STUDY

Another interesting application of Scintillae concerns cryptography where cellular automata revealed to be very useful [8, 18, 17]. It is very easy to construct in Scintillae symmetric key systems to encode and decode sequences of bits. In this field, a typical method to encode a message is to consider its representation as a sequence of bits, and performing a bitwise XOR with another pseudo-random sequence of bits of the same length. In this way, an encrypted message is obtained. To decode the encoded message, it is sufficient to repeat the bitwise XOR with the same pseudo-random sequence of bits. The knowledge needed to reproduce the pseudo-random sequence of bits to both the sender and the receiver represents the symmetric key. An implementation of such systems in Scintillae is shown in Figure 15. The circuit included in the rectangle denoted as “Bits generator” is the core of the system as it produces the required pseudo-random sequence of bits encoded as sparks outgoing from its top rightmost arrow. The rational behind it is to have a set

---

**FIGURE 14**
The seven-bars display, the activation functions of its input lines, and the circuit used to convert a binary string of two bits XY into the appropriate signals.
of arrows and sparks randomly disposed in order to obtain a random sequence of sparks as output. However, it is also useful to easily reproduce the obtained disposal as this represents the symmetric key necessary to encode and decode the message. That’s why in our implementation, the Bits generator is made of lines of arrows such that two consecutive lines point in opposite directions. In the figure, odd lines point to the right, even lines point to the left. In doing so, a spark carried on an even line might be reproduced by the arrow on its top or bottom of the neighboring odd lines. However, since an arrow might be neighbor of two sparks from two lines of the same type, then in such a case it does not output a spark. In order to brake the symmetry of the circuit, one further arrow on the bottom-left part of the circuit has been added.

The sequence produced by the Bits generator along with the “Message” sequence is given in input to the arrow denoted by “XOR” in the Figure. On the output line starting from the XOR, the “Encoded Message” is produced. In order to delimit the beginning and the end of a message, the circuit makes use of two sparks on the “Start/End delimiters” line. These two sparks are also used to open and close the “tap” T1. In fact, T1 is open only when a message traverses the XOR gate. The randomness of the sequence of bits provided by the Bits generator circuit relies on the initial disposal of its arrows and sparks. In our example, the period after which the string repeats is of 176 clock’s ticks even though the circuit is very small. Such a circuit can be enlarged as desired in order to obtain much longer periods. Another way to produce longer sequences can be obtained by combining the outputs of two
Bits generators as the input of an arrow. This realizes the XOR of the input sequences, and by [12], the resulting sequence has a period equal to the least common multiple of the periods of the two input sequences if, e.g., they are coprime. More specifically:

**Theorem 1** [12] Let \( a = (a_0, a_1, \ldots) \) be a periodic sequence of (minimal) period \( n \), and let \( b = (b_0, b_1, \ldots) \) be a periodic sequence of (minimal) period \( m \). Let \( c = (c_0, c_1, \ldots) \) be the sequence with \( c_i = a_i \ XOR \ b_i \) for each \( i \). Suppose that for every prime \( r \), the largest power of \( r \) that divides \( n \) is not equal to the largest power of \( r \) that divides \( m \). Then \( c \) is periodic and the period of \( c \) is the least common multiple of \( n \) and \( m \).

The symmetric key \( K^S(t_0) \) of the circuit is represented by the disposal of the arrows along with a specific disposal of sparks on the Bits generator which provides its status \( S \) at the starting time \( t_0 \). Let \( K^S(t) \) be the status of the Bits generator after \( t \) clock’s ticks from \( t_0 \). In order to decode an encoded message, the same circuit is required. We denote by \( K^S_{\text{e}} \) and \( K^S_{\text{d}} \) the status of the encoder and decoder, respectively. Of course, encoder and decoder circuits must be synchronized: The status of the encoder after \( t \) clock’s ticks, \( K^S_{\text{e}}(t) \), must be equal to that of the decoder after \( n \) clock’s ticks, i.e. \( K^S_{\text{d}}(t + n) \), with \( n \) being the clock’s ticks required for the message to reach the decoder from the encoder.

An interesting study is the relation between the size of the Bits generator and the period of the sequence of bits generated as well as to measure the randomness of such sequences.

6 CONCLUSION

We have presented Scintillae, a new and simple cellular automaton that reveals high computational power capabilities. Surprising results have been obtained by suitably placing the few pieces provided by the game. Educational characteristics could also be exploited. In fact, Scintillae turns out to be a very good mean for experiencing sequential circuits, transmission systems, cryptography, and also for an easy approach to low level programming languages like assembly.

7 HISTORICAL NOTE AND ACKNOWLEDGEMENT

A first version of Scintillae has been implemented by Gabriele Di Stefano on a PC Olivetti M24 in the mid-eighties.
Special thanks go to Gian Marco Tedesco for his great contribution in coding the first version of Scintillae with graphic libraries, for his insights and useful discussions.

Recently, a new implementation has been released by Mirco Tracolli as part of his BSc thesis [3]. Figures 13 and 15 show the new environment. The thesis addressed some of the future works posed in the conference paper [10]. These include multi-platform and open sources implementation plus modularity. One of the main useful advances allows to exploit configurations already defined as black boxes for new configurations. This expands the visual programming features of the game and permits to simplify big projects.

REFERENCES