3.2.3 Von Neumann's Successors

The extreme size of von Neumann's universal constructor has so far prevented any kind of physical implementation (apart from the small demonstration unit we mentioned). But further, even the simulation of a cellular automaton of such complexity was far beyond the capability of early computer systems. Today, such a simulation is starting to be conceivable. Umberto Pesavento, a young Italian high school student, developed a simulation of von Neumann's entire universal constructor [78]. The computing power available did not allow him to simulate either the entire self-replication process (the length of the memory tape needed to describe the automaton would have required too large an array) or the Turing machine necessary to implement the universal computer, but he was able to demonstrate the full functionality of the constructor. Considering the rapid advances in computing power of modern computer systems, we can assume that a complete simulation could be envisaged within a few years.

The impossibility of achieving a physical realization did not however deter some researchers from trying to continue and improve von Neumann's work [11, 54, 71]. Arthur Burks, for example, in addition to editing von Neumann's work on self-replication [17, 104], also made several corrections and advances in the implementation of the cellular model. Codd [20], by altering the states and the transition rules, managed to simplify the constructor by a considerable degree. However, without in any way lessening these contributions, we can say that no major theoretical advance in the research on self-reproducing automata occurred until C. Langton, in 1984, opened a second stage in this field of research.

3.3 Langton's Loop

Von Neumann's Universal Constructor was so complex because it tried to implement self-reproduction as a particular case of construction universality, i.e. the capability of constructing any other automaton, given its description. C. Langton approached the problem somewhat differently, by attempting to define the simplest cellular automaton capable exclusively of self-reproduction.

As a consequence of this approach, his automaton, commonly known as *Langton's Loop* [53], is orders of magnitude simpler than von Neumann's. In fact, it is loosely based on one of the simplest organs[9] in Codd's automaton: the periodic emitter (itself derived from von Neumann's periodic pulser), a relatively simple structure capable of generating a repeating string of a given sequence of pulses.

Langton's loop (Fig. 3-6) is named after the dynamic storage of data inside a square sheath (red in the figure). The data is stored as a sequence of instructions for directing the constructing arm, coded in the form of a set of three states. The data turns counterclockwise in permanence within the sheath, creating a loop.

9. An organ in this context can be seen as a self-supporting structure capable of a single sub-task.

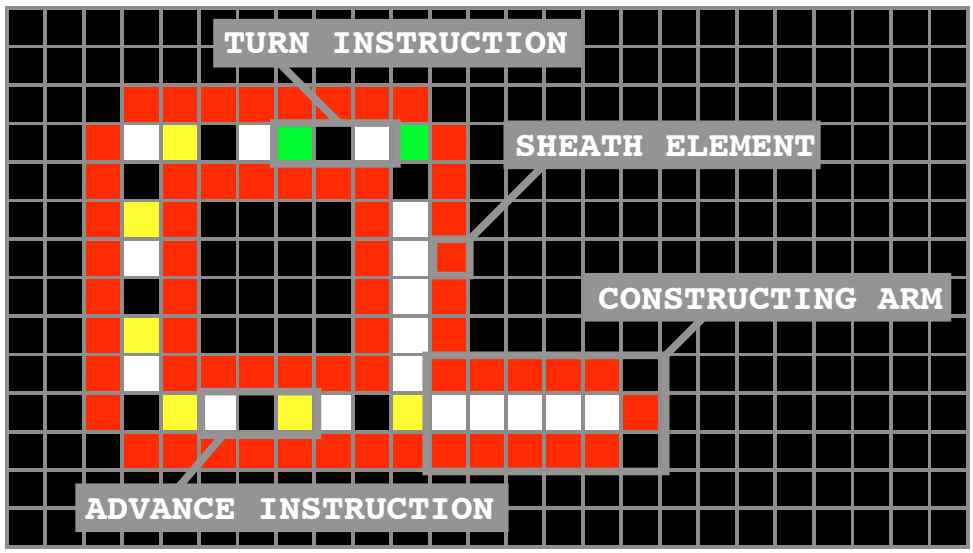

Figure 3-6: The initial configuration of Langton's Loop (iteration 0).

The two instructions in Langton's loop are extremely simple. One instruction (uniquely identified by the yellow element in the figure) tells the arm to advance by one position (Fig. 3-7), while the other (green in the figure) directs the arm to turn 90° to the left (Fig. 3-8). Obviously, after three such turns, the arm has looped back on itself (Fig. 3-9), at which stage a messenger (the pink element) starts the process of severing the connection between the parent and the offspring, thus concluding the replication process. Once the copy is over, the parent loop proceeds to construct a second copy of itself in a different direction (to the north in this example), while the offspring itself starts to reproduce (to the east in this example).

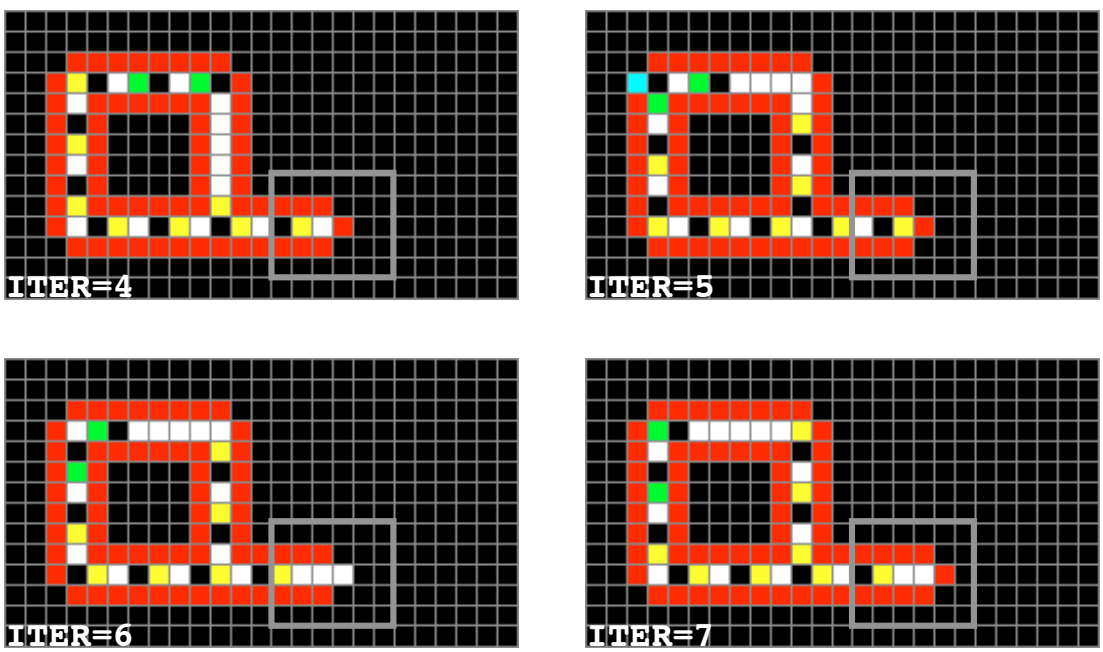

Figure 3-7: The constructing arm advances by one space.

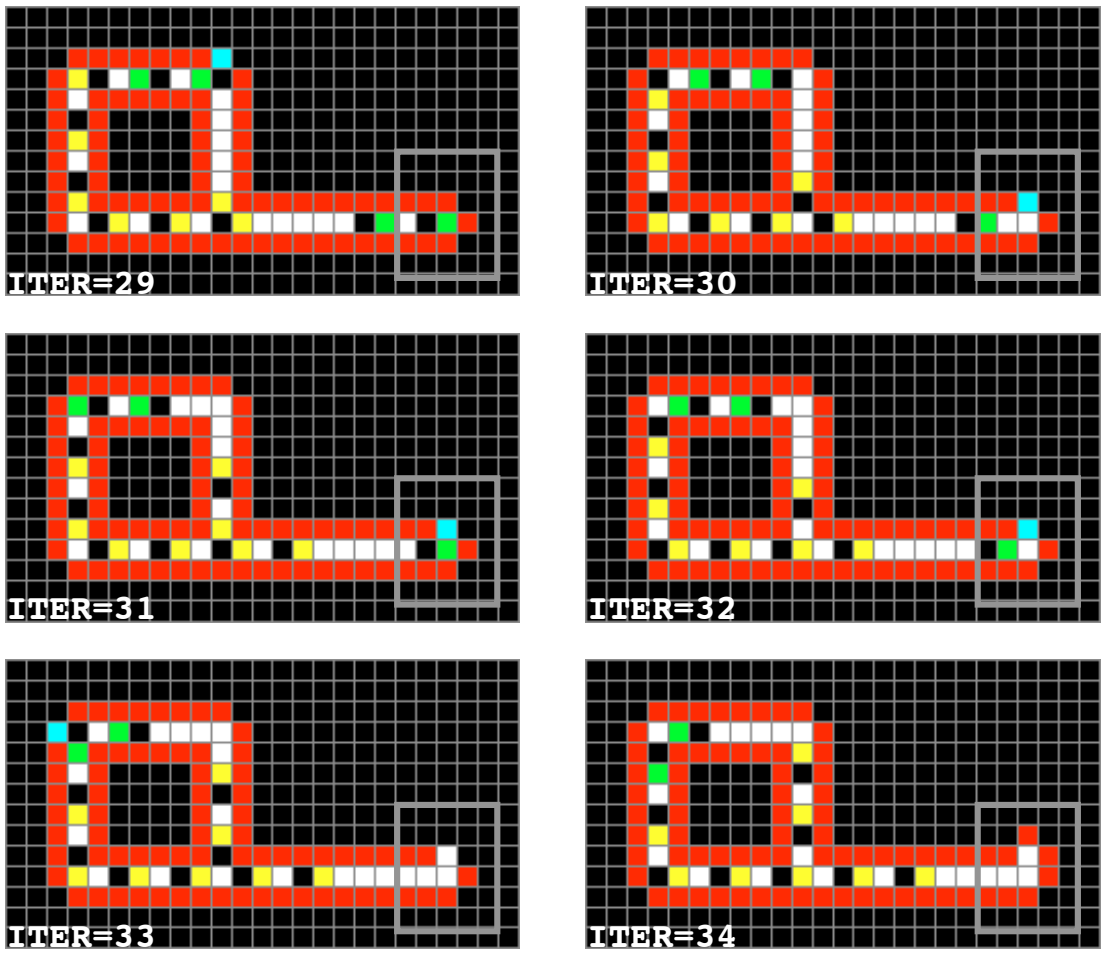

Figure 3-8: The constructing arm turns 90º to the left.

The sequential nature of the self-reproduction process generates a spiraling pattern in the propagation of the loop through space (Fig. 3-10): as each loop tries to reproduce in the four cardinal directions, it finds the place already occupied either by its parent or by the offspring of another loop, in which case it dies (the data within the loop is destroyed).

Langton's loop uses 8 states for each of the 86 non-quiescent cells making up its initial configuration, a 5-cell neighborhood, and a few hundred transition rules (the exact number depends on whether default rules are used and whether symmetric rules are included in the count).

Further simplifications to Langton's automaton were introduced by Byl [18], who eliminated the internal sheath and reduced the number of states per cell, the number of transition rules, and the number of non-quiescent cells in the initial configuration. Reggia et al. [82] managed to remove also the external sheath, thus designing the smallest self-replicating loop known to date. Given their modest complexity, at least relative to von Neumann's automaton, all of the mentioned automata have been thoroughly simulated.

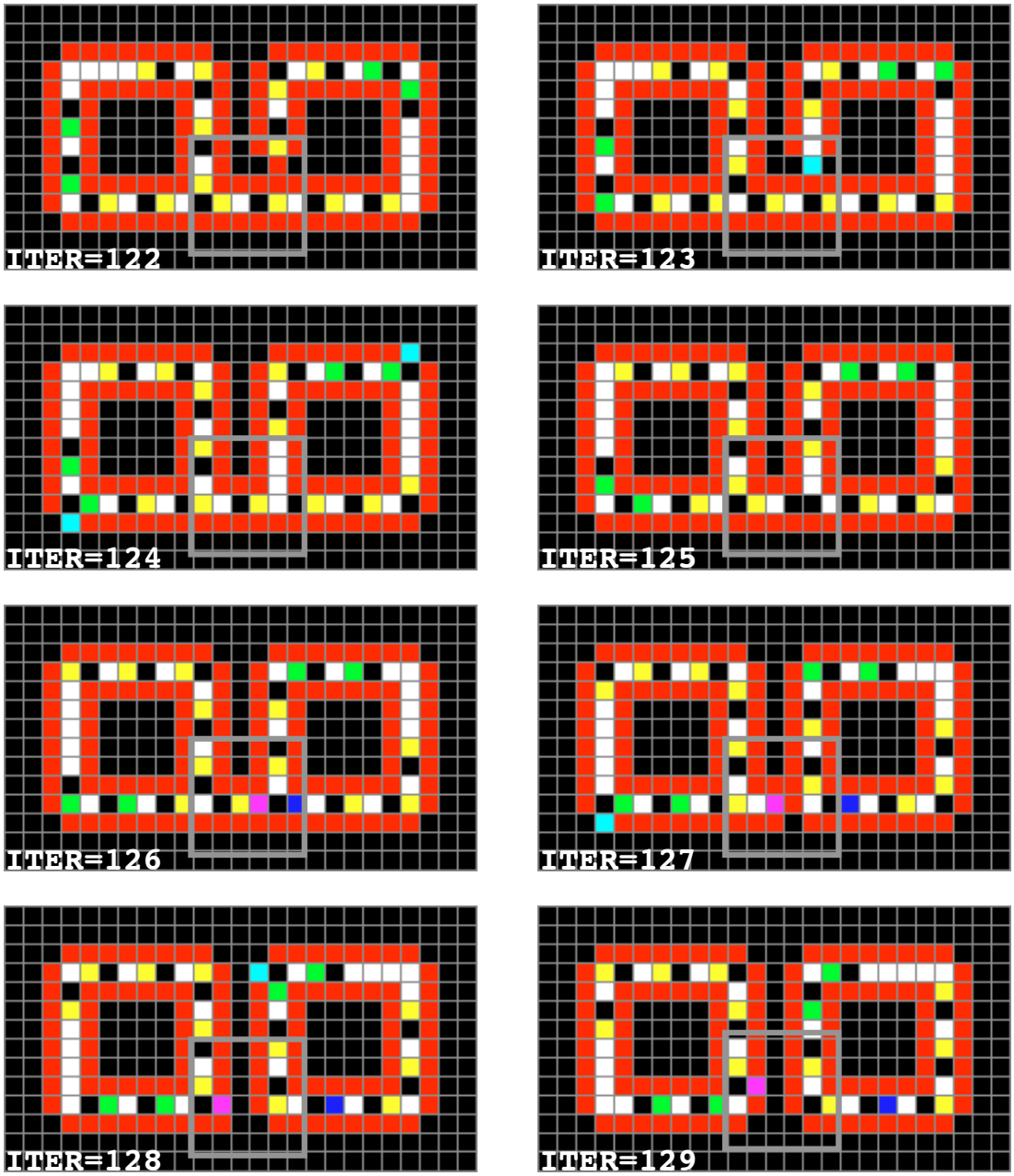

Figure 3-9: The copy is complete and the connection from parent to offspring is severed.

3.4 Self-Replicating Cellular Automata in Embryonics

While the self-replicating automata we introduced in this chapter are indeed interesting examples of self-replicating machines, they do not address some of the requirements of the Embryonics project. In this section, we will attempt to define more precisely these requirements and, after introducing a first attempt to augment the versatility of Langton's loop through the addition of a Turing machine, we will present a new self-replicating automaton we developed in order to offset

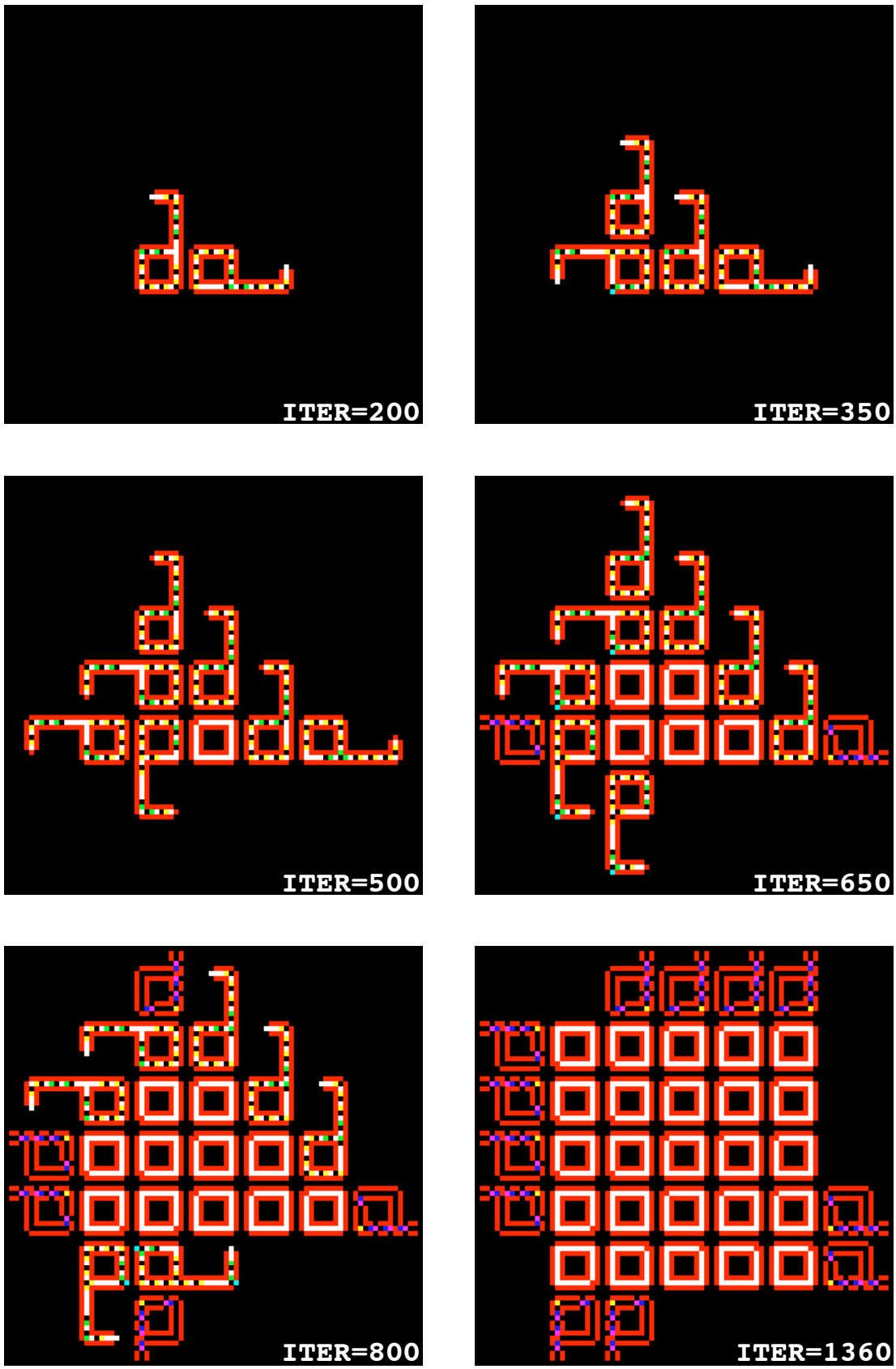

Figure 3-10: Propagation pattern of Langton's loop.

some of the more important deficiencies. We will begin by describing in some detail the operation of the new loop in its most basic configuration, and then illustrate an example of a more elaborate version where a program has been added to the basic automaton to demonstrate its construction capabilities.

3.4.1 The Requirements of the Embryonics Project

Langton's loop represents the best-known example of a simple self-replicating machine, and is therefore of great interest for the Embryonics project. However, it falls short of our requirements in two important aspects:

1. It is designed to operate in an infinite space, whereas the surface of an integrated circuit is necessarily finite. This inconvenience, which might appear relatively minor and easily circumvented, is in fact a major obstacle: the loop's mechanism of self-replication (i.e. the arm) is inherently incapable of handling contact with the array's border.
2. It does not have any functionality beyond self-replication: the loop reproduces and then dies. It is thus more similar to a biological (or even a software) virus than to an actual cell. Once again, the default is structural: because of its origins in the periodic emitter, all the data circulating inside the loop is involved in the generation of the sequence which directs the self-replication process. As we will see, while it is possible to add functionality to Langton's loop, the task is extremely complex and the results not very efficient.

Nevertheless, von Neumann's constructor and Langton's loop are the models which most closely approach our requirements. Therefore, we decided to follow tradition in developing our own approach to self-replication by using cellular automata as an environment to study the issue.

As we already mentioned, there is no formal approach to the development of complex cellular automata. The design of such systems poses therefore considerable problems, since few of the available tools are suited for the task. In particular, no efficient tools were available to help the user in determining the local transition rules necessary to obtain a complex global behavior (cellular automata are mostly used to simulate physical phenomena, where the rules are usually well known). Our first task was therefore to design a software application which would allow us to generate the required rules as easily as possible. The resulting program (described in Appendix A) is, to the best of our knowledge, unique, and proved an invaluable tool in our research.

Equipped with a reasonably powerful tool, we started developing a new automaton capable of self-replication. Considering the complexity of von Neumann's constructor, we decided to draw inspiration from the much simpler Langton's loop, but, as we will see, we had to develop a completely novel mechanism to enable us to circumvent its drawbacks.

3.4.2 A Self-Replicating Turing Machine: Perrier's Loop

As we mentioned, one of the two main problems of Langton's loop is that it is not well adapted to finite CA arrays. Its self-reproduction mechanism assumes that there is enough space for a copy of the loop, and the entire loop becomes non-functional otherwise.

At first glance, this might seem a relatively trivial drawback, which could be overcome by modifying the loop. Such a modification, however, turns out to be very difficult. In fact, to exist in a finite space, and assuming that the automaton has no *a priori* knowledge of the location of the boundaries (a safe assumption, since CA elements have only knowledge of their immediate neighborhood), the loop needs either to verify that enough space is available before it starts the replication process, or else some way to destroy the constructing arm if it detects a boundary during the self-replication process. Either of these mechanisms would require a major structural modification to Langton's loop.

Thus, rather than trying to adapt Langton's automaton to a finite space, we decided to develop an entirely new mechanism, designed specifically to exist in a finite, but arbitrarily large, space, and at the same time capable, unlike Langton's loop, to have a functionality in addition to self-replication.

Adding functionality to Langton's loop, in fact, is not possible without major alterations. As a matter of fact, in the course of our research, we did develop a relatively complex automaton (Fig. 3-11) in which a two-tape Turing machine was appended to Langton's loop [77]. This automaton, developed in our laboratory by

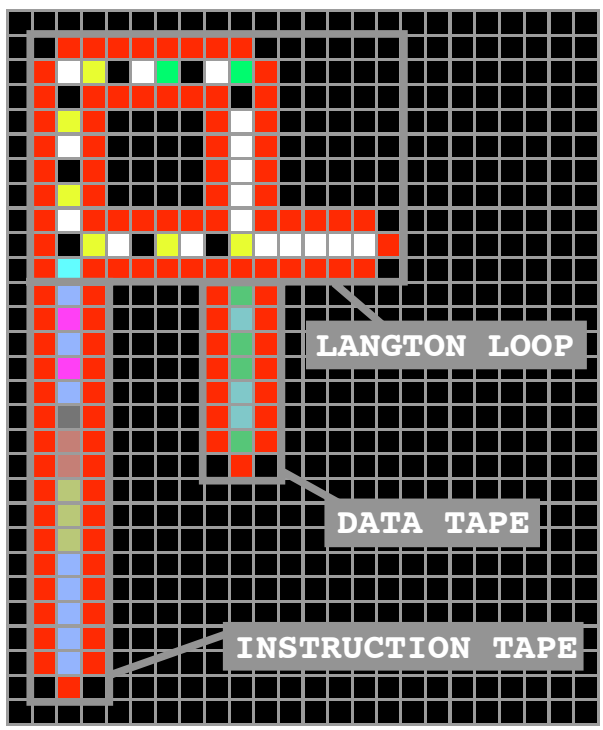

Figure 3-11: A two-tape Turing machine appended to Langton's loop (iteration 0).

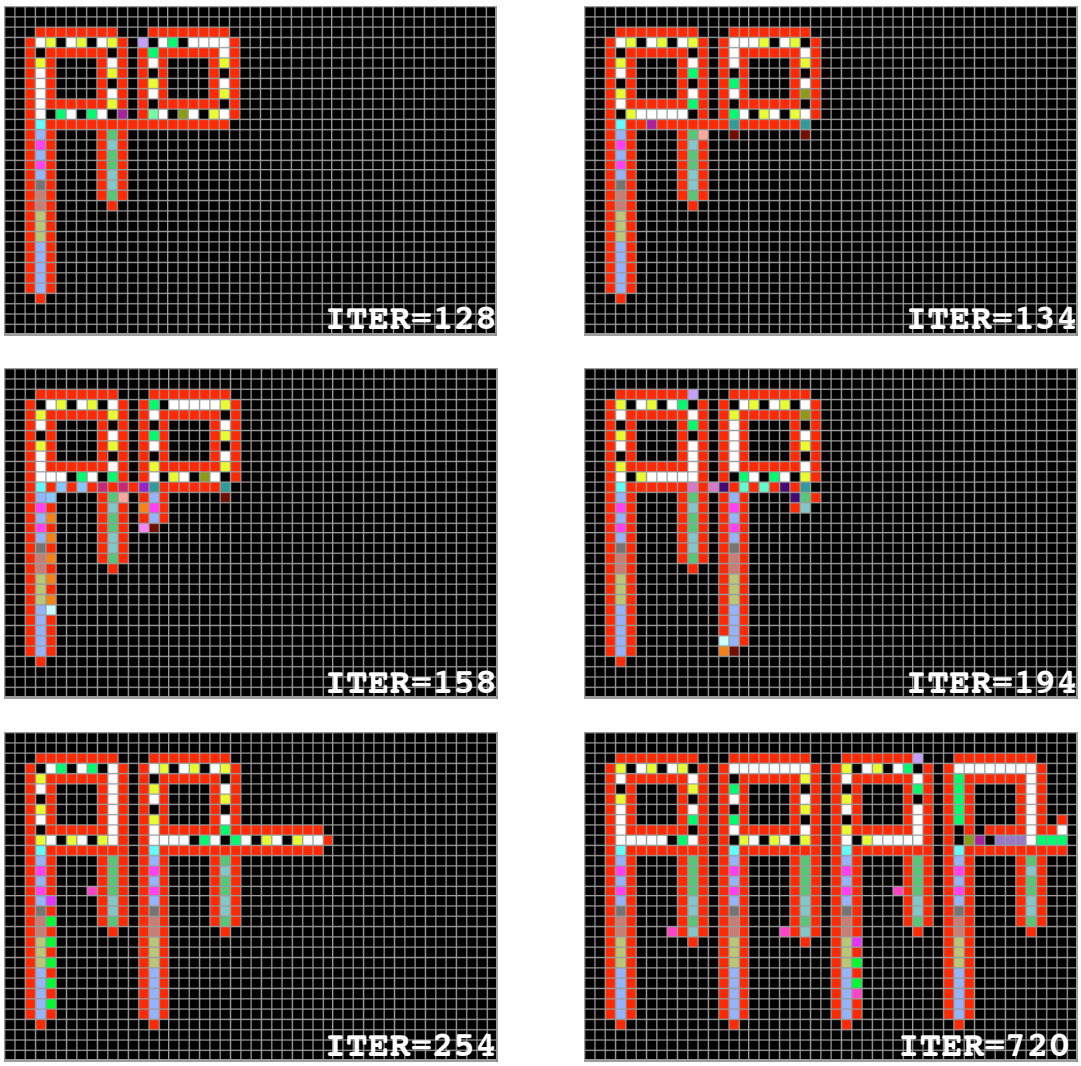

Figure 3-12: Self-replication of the Turing machine.

J.Y. Perrier as a semester-long research project under the supervision of Prof. J. Zahnd, exploits Langton's loop as a sort of "carrier" (Fig. 3-12): the first operation of *Perrier's loop* is to allow Langton's loop to build a copy of itself (iteration 128: note that the copy is limited to one dimension, since the second dimension is taken up by the Turing machine). The main function of the offspring is to determine the location of the copy of the Turing machine (iteration 134). Once the new loop is ready, a "messenger" runs back to the parent loop and starts to duplicate the Turing machine (iterations 158 and 194), a process completely disjoint from the operation of the loop. When the copy is finished (iteration 254), the same messenger activates the Turing machine in the parent loop (the machine had to be inert during the replication process in order to obtain a perfect copy). The process is then repeated in each offspring until the space is filled (iteration 720: as the automaton exploits Langton's loop for replication, meeting the boundary of the array causes the last machine to crash).

The automaton thus becomes a self-replicating Turing machine, a powerful construct which is unfortunately handicapped by its complexity: in order to implement a Turing machine, the automaton requires a very considerable number of additional states (more than 60), as well as an important number of additional transition rules. This kind of complexity, while still relatively minor compared to von Neumann's universal constructor, is nevertheless too important to be really considered for a hardware application. So once again, adapting Langton's loop to fit our requirements proved too complex to be efficient, and we were forced to design a novel automaton to meet our requirements.

3.4.3 A Novel Self-Replicating Loop: Description

In designing our self-replicating automaton [97], we did maintain some of the more interesting features of Langton's loop. In particular, we preserved the structure based on a square loop to dynamically store information. Such storage is convenient in CA because of the locality of the rules. Also, we maintained the concept of constructing arm, in the tradition of von Neumann and his successors, even if we introduced considerable modifications to its structure and operation.

While preserving some of the more interesting features of Langton's loop, we nevertheless introduced some basic structural alterations:

- We use a 9-element neighborhood (the element itself plus its 8 neighbors).
- As in Byl's version of Langton's loop, we use only one sheath, but contrary to Byl, we retain the internal sheath and eliminate the external one. This allows us to let the data in the loop circulate without the need for leading or trailing states (the black and white elements in Langton's loop). In addition to the internal sheath, we have four *gate elements* (in the same state as the sheath) outside the loop at the four corners of the automaton. These elements are initially in the "open" position, and will shift to the "closed" position once the copy is accomplished.
- We extend four constructing arms in the four cardinal directions at the same time, and thus create four copies of the original automaton in the four directions in parallel. When the arm meets an obstacle (either the boundary of the array or an existing copy of the loop), it simply retracts and puts the corresponding gate element in the closed position. This mechanism allows us to overcome the first major drawback of Langton's loop in relation to the Embryonics project (its inability to work properly in a finite space).
- Rather than being directed to advance, our constructing arm advances by default. As a consequence, it is necessary only to direct it to turn at the appropriate moment. This is done by sending periodic "messengers" to the tip of the constructing arm.

- The arm does not immediately construct the entire loop. Rather, it constructs a sheath of the same size as the original. Once the sheath is ready, the data circulating in the loop is duplicated and the copy is sent along the constructing arm to wrap around the new sheath. When the new loop is completed, the constructing arm retracts and closes the gate. As we will see, dividing the self-replication process in two phases is a major asset in the transition to digital hardware.
- As a consequence, we use only four of the circulating elements to generate the messengers. Since the only operation performed on the remaining data elements is duplication, they do not have to be in any particular state. In particular, they can be used as a "program", i.e., a set of states with their own transition rules which will then be applied alongside the self-reproduction to execute some function, allowing us to overcome the second drawback of Langton's loop (its lack of functionality beyond self-reproduction).
- Unlike Langton's loop, our loop does not "die" once duplication is achieved, as the circulating data remains untouched by the self-reproduction process. This feature is a requirement for implementing functions which work after the copy has finished. As a side benefit, it becomes possible to force the loop to try and duplicate again in any of the four directions simply by shifting the corresponding gate back to the open position. This feature could be interesting in view of self-repair: a dead loop can be reconstructed by its neighbors.

As should be obvious, while our loop owes to von Neumann the concept of constructing arm and to Langton (and/or Codd) the basic loop structure, it is in fact a very different automaton, endowed with some of the properties of both.

We have seen that von Neumann's automaton is extremely complex, while Langton's loop is very simple. The complexity of our automaton is more difficult to estimate, as it depends on the data circulating in the loop. The number of non-quiescent elements making up the initial configuration depends directly on the size of the circulating program. The more complex (i.e. the longer) the program, the larger the automaton (it should be noted, however, that the complexity of the self-reproduction process does not depend on the size of the loop). The number of states also depends on the complexity of the program. To the 5 "basic" states used for self-reproduction (see description below) must be added the "data states" (at least one) used in the program, which must be disjoint from the basic states. The number of transition rules is obviously a function of the number of data states: in the basic configuration (i.e., one data state), the automaton needs 692 rules[10] (173 rules rotated in the four directions).

The complexity of the basic configuration is therefore in the same order as that of Langton's and Byl's loops, with the proviso that it is likely to increase drastically if the data in the loop is used to implement a complex function.

10. By default, all elements remain in the same state.

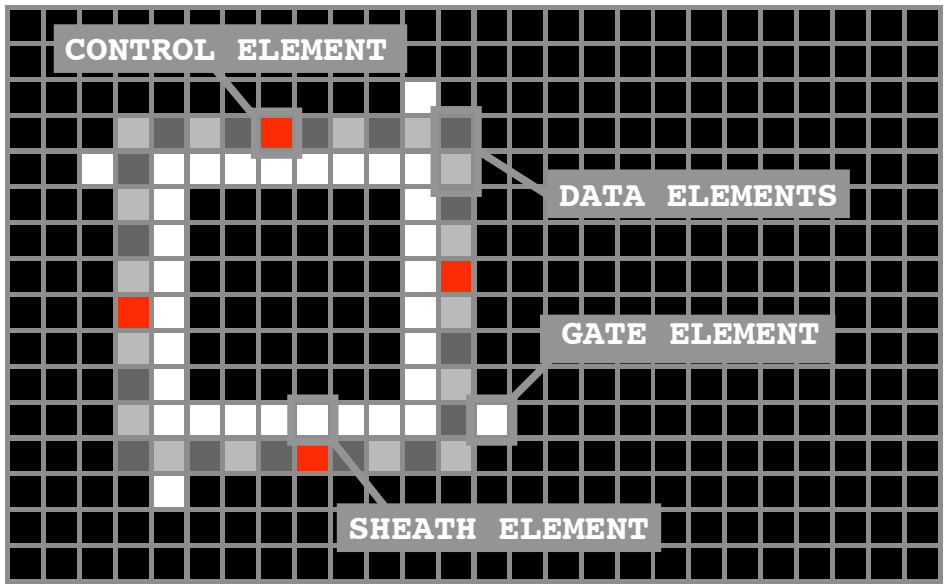

Figure 3-13: The initial configuration of the loop (iteration 0).

3.4.4 A Novel Self-Replicating Loop: Operation

As for von Neumann's and Langton's automata, the ideal space for our automaton is an infinite two-dimensional grid. Since we realize that a practical implementation of such a space might prove difficult, we added some transition rules to handle the collision between the constructing arm and the border of the array. On meeting the border, the arm will retract without attempting to make a copy of the parent loop.

The elements of the array require five basic states and at least one data state (Fig. 3-13). State 0 (black) is the *quiescent state*: it represents the inactive background. State 1 (white) is the *sheath state*, that is the state of the elements making up the sheath and the four gates. State 2 (red) is the *activation state* or *control*

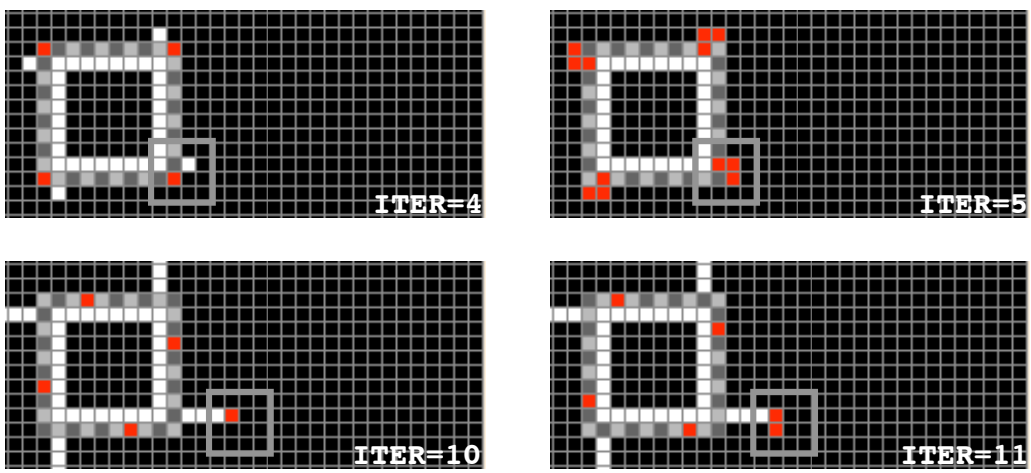

Figure 3-14: The constructing arm begins to extend.

state. The four elements in the loop directing the reproduction are in state 2, as are the messengers which will be used to command the constructing arm and the tip of the constructing arm itself for the first phase of construction, after which the tip of the arm will pass to state 3 (light blue), the *construction state*. State 3 will construct the sheath that will host the offspring, signal the parent loop that the sheath is ready, and lead the duplicated data to the new loop. State 4 (green), the *destruction state*, will destroy the constructing arm once the copy is completed. In addition to these states, two additional *data states* (light and dark grey) represent the information stored in the loop. In this example, they are inactive, while the next section describes a loop where they are used to store an executable program.

The initial configuration is in the form of a square loop wrapped around a sheath. The size of the loop is variable, and for our example is set to 8x8. In the loop is a string of elements of which four are in the activation state (red) and are placed at a distance from each other equal to the side of the loop. Near the four corners of the loop we have placed four elements in the sheath state. These are the gate elements, and the position they occupy at iteration 0 means that the gates are open (that is, that the automaton should attempt to duplicate itself in all four directions).

Once the iterations begin, the data starts turning counterclockwise around the loop. Nothing happens until the first control element reaches a corner of the loop, where it checks the status of the gate. Since the gate is open, the control element splits into two identical elements: the first continues turning around the loop, while the second starts extending the arm (Fig. 3-14). The arm advances by one position every two iterations. Once the arm has started extending, each control element that arrives to a corner will again split and one of the copies will start running along the arm, advancing by one position per iteration (Fig. 3-15). Since the arm is extending at half the speed of the messengers and the messengers are spaced 8 elements apart (the length of one side of the loop), the messengers will reach the tip of the arm at regular intervals corresponding to the length of one side of the loop.

When the first messenger reaches the tip of the arm, the tip, which was until then in state 2, passes to state 3 and continues to advance at the same speed (Fig. 3-16). This transformation tells the arm that it has reached the location of the offspring loop and to start constructing the new sheath.

The next three messengers will force the tip of the arm to turn left (Fig. 3-17), while the fourth will reach the tip as the arm is closing upon itself (Fig. 3-19). It causes the sheath to close and then runs back along the arm to signal to the original loop that the new sheath is ready.

Once the return signal arrives at the corner of the original loop, it waits for the next control element to arrive (Fig. 3-20). When the control element sees the messenger waiting by the gate, once again it splits, one copy staying around the

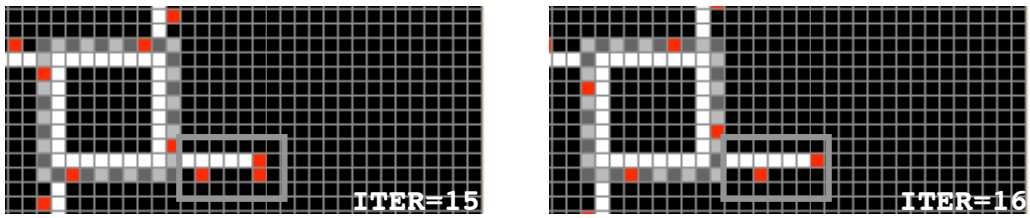

Figure 3-15: The first messenger is running along the arm.

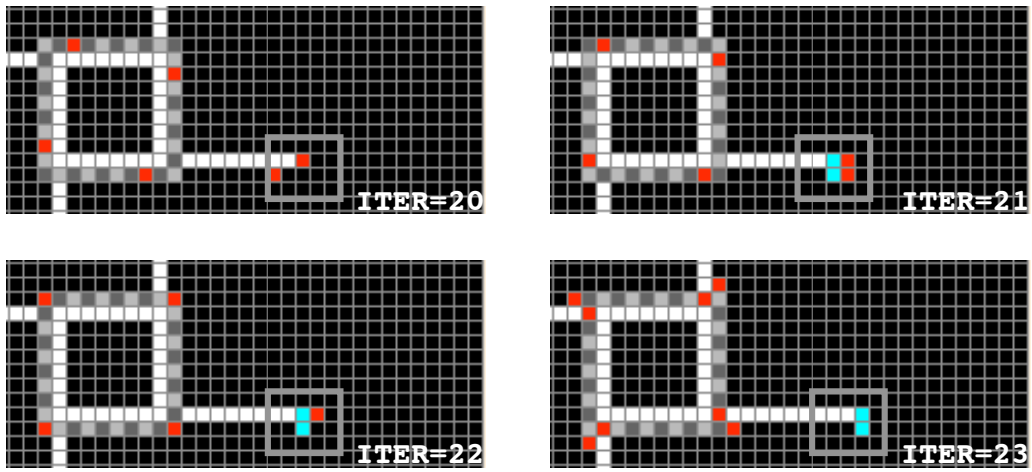

Figure 3-16: The first messenger reaches the tip of the constructing arm.

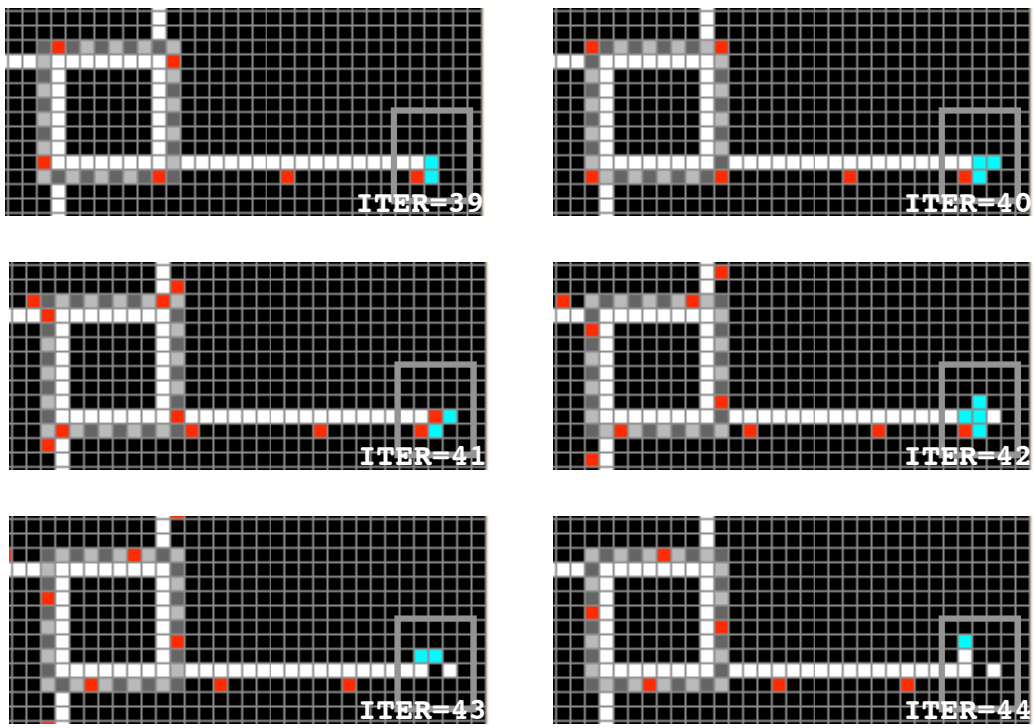

Figure 3-17: The second messenger reaches the tip of the arm, forcing it to turn left.

loop, the other running along the arm. This time, however, rather than running along the arm in isolation as a messenger, it carries behind him a copy of the data in the loop.

Always followed by the data, it runs around the sheath until it has reached the junction where the arm folded upon itself (Fig. 3-18). On reaching that spot, it closes the loop and sends a destruction signal (green) back along the arm. The signal will destroy the arm until it reaches the corner of the original loop, where it closes the gate to avoid further copies.

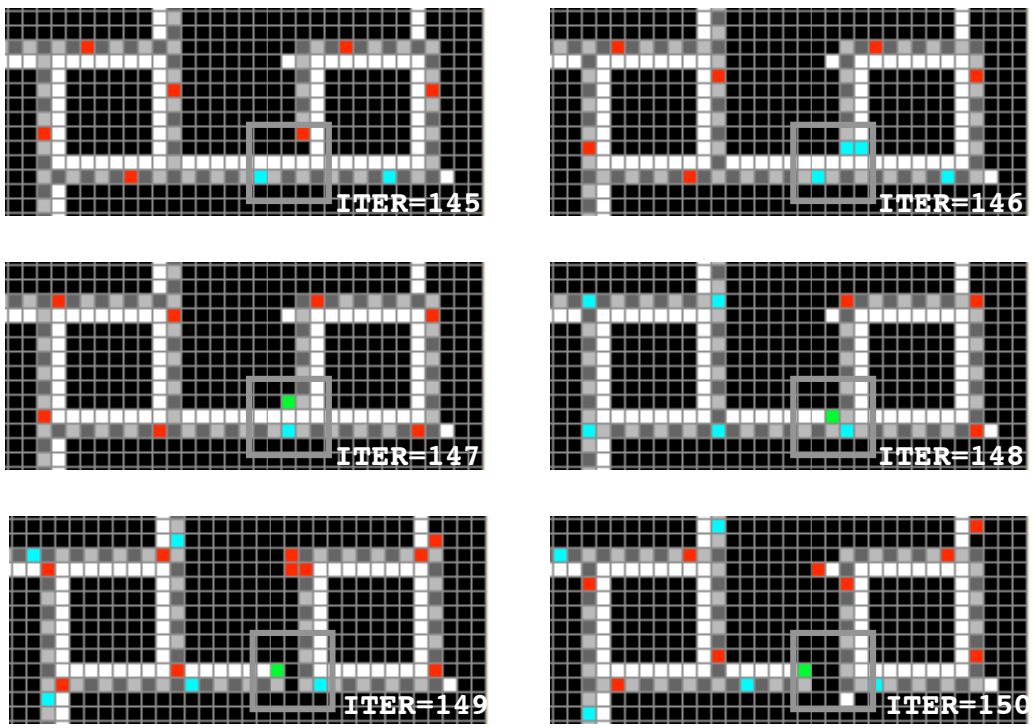

Figure 3-18: The copy is complete and the constructing arm retracts.

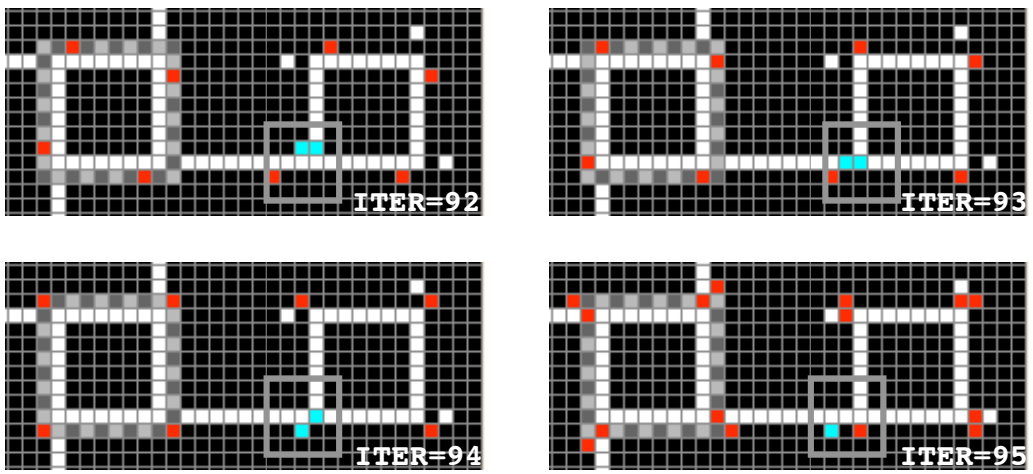

Figure 3-19: The loop is closed, and a new messenger (light blue) is sent back to the parent loop to signal that the offspring is ready to receive the data.

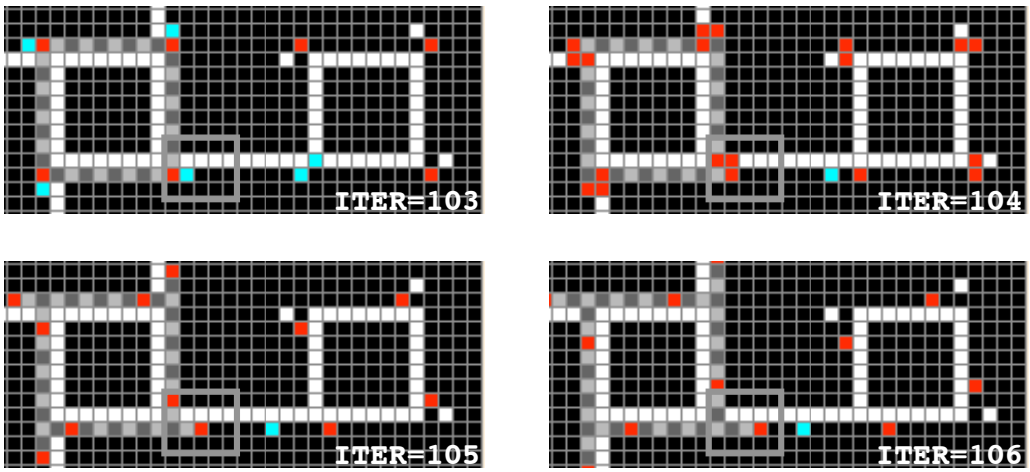

Figure 3-20: A copy of the data is sent from the parent to the offspring.

Meanwhile, the new loop is already staring to reproduce itself in three of the four directions. One direction (down in the figures) is not necessary since another of the new loops will always get there first, and therefore its corresponding gate is automatically set to the closed position. Since the automaton reproduces in all four directions at the same time, its propagation pattern (Fig. 3-21) is somewhat different from that of Langton's loop.

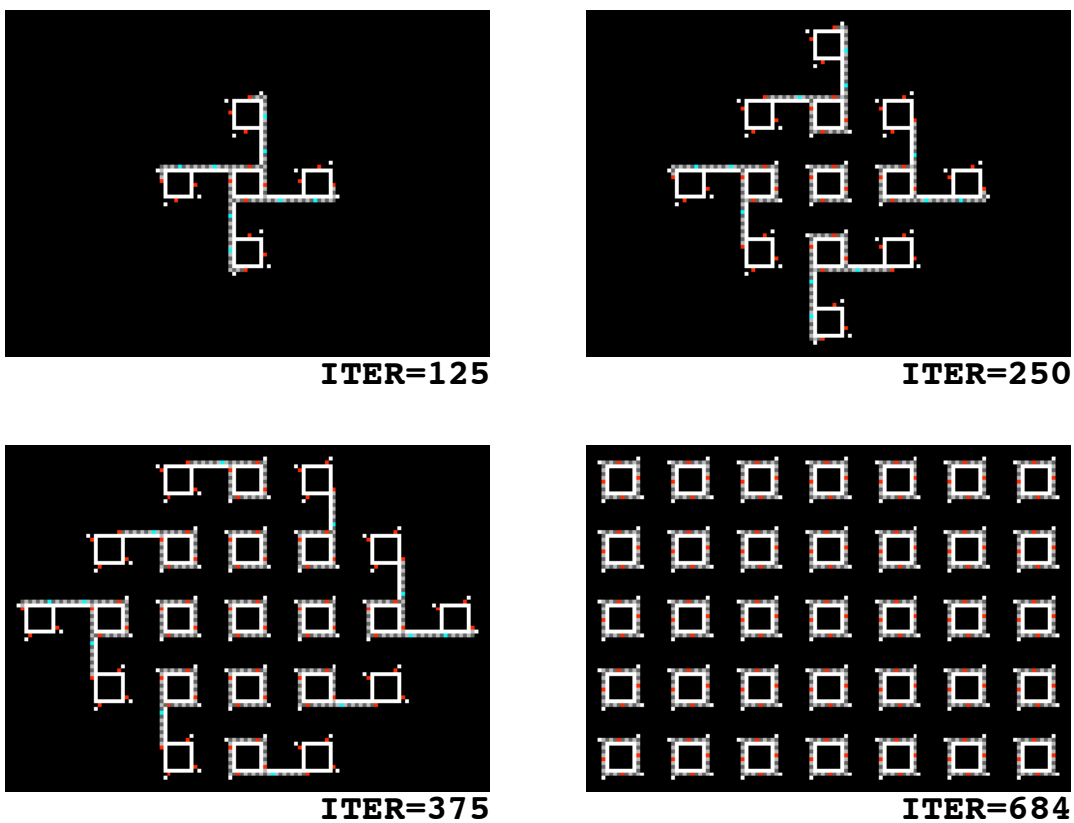

Figure 3-21: The propagation pattern for our loop.

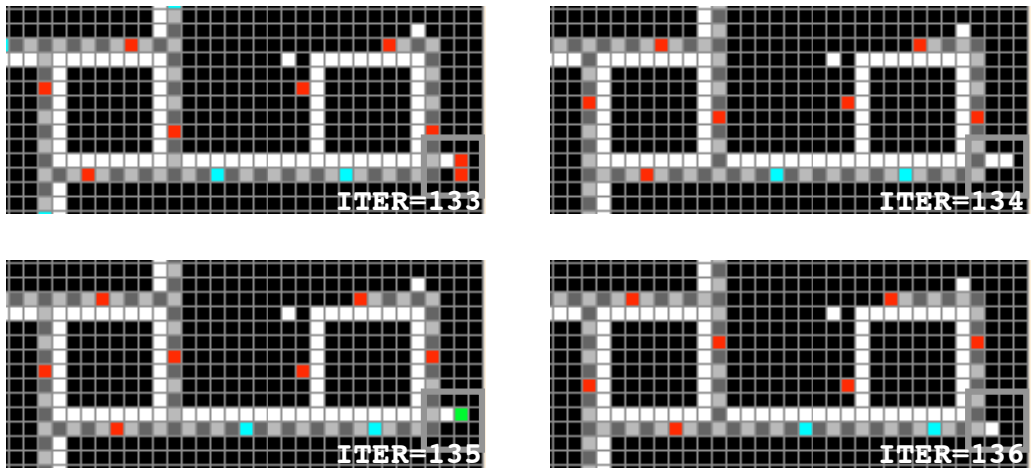

Figure 3-22: The arm, finding the boundary of the array, retracts and closes the gate.

After 121 time periods the gates of the original automaton will be closed and it will enter an inactive state, with the understanding that it will be ready to reproduce itself again should the gates be opened.

The main advantage of the new mechanism is that it becomes relatively simple to retract the arm if an obstacle (either the boundary of the array or another loop) is encountered, and therefore our loop is perfectly capable of operating in a finite space. In the example above, the right border of the figure corresponded to the boundary of the array: when the offspring tried to replicate towards the east, the arm, it found its way blocked, simply retracted and closed its gate (Fig. 3-22).

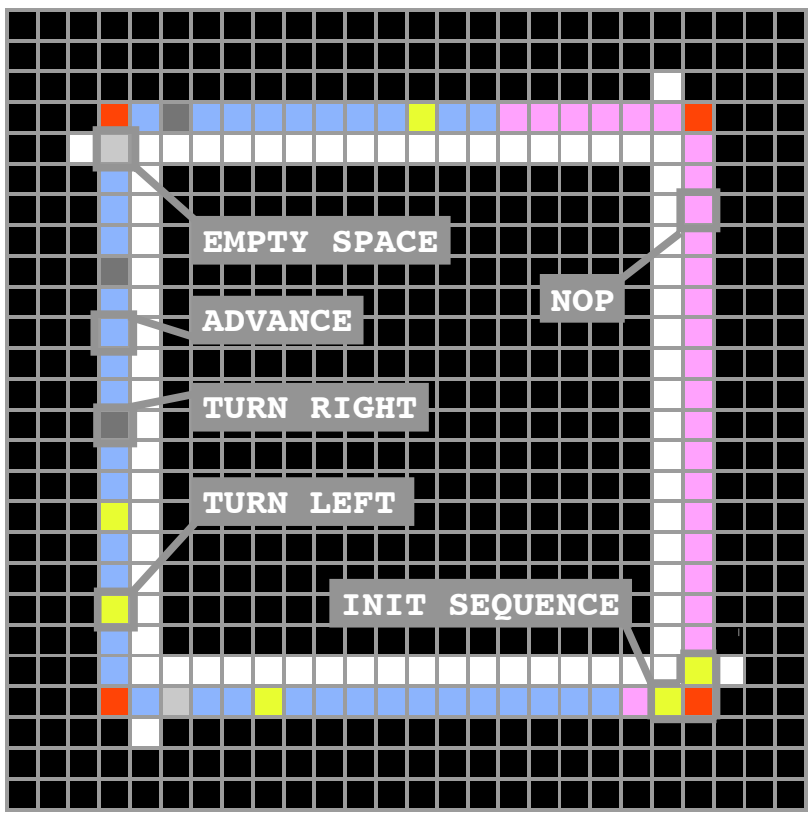

Figure 3-23: Configuration of the LSL automaton at iteration 0.

3.4.5 A Novel Self-Replicating Loop: a Functional Example

In Fig. 3-23, we illustrate an example of how the data states can be used to carry out operations alongside self-reproduction. The operation in question is the construction of three letters, LSL (the acronym of Logic Systems Laboratory), in the empty space inside the loop. Obviously this is not a very useful operation from a computational point of view, but it is a far from trivial construction task which should suffice to demonstrate the capabilities of the automaton.

For this example, we have used 5 data states, each representing an instruction for the construction of the letters: *advance*, *turn left*, *turn right*, *empty space*, and a NOP (no operation) instruction to fill the remaining space in the loop. The construction requires 330 additional rules.

The operation of the program is fairly straightforward. When a certain *initiation sequence* within the loop arrives to the top left corner of the loop, a "door" is opened in the internal sheath (Fig. 3-24). The rest of the program, as it passes by the door in its rotation around the loop, is duplicated and one of the copies is sent to the interior of the loop, where it is treated as a sequence of instructions which direct the construction of the three letters. Once the duplication is complete (i.e., when the first NOP instruction reaches the opening), the door is closed and the sheath reset, except for a flag which indicates that the task has already been completed and prevents the door from being opened again (Fig. 3-25).

The construction mechanism itself is somewhat similar to the method Langton used in his own loop, and is based on a modified constructing arm. The advance instruction causes the arm to advance by one element, the turn left and turn right (Fig. 3-26) instructions cause the arm to change direction, and the "empty space" instruction produce a gap in the arm (to separate the letters).

During the process of reproduction, the program is simply copied (as opposed to interpreted as in the interior of the sheath) and arrives intact in the new loop, where it will execute again exactly as it did in the parent loop (Fig. 3-27).

This is a simple demonstration of one way in which the data in the loop could be used as an executable program. Of course, many other methods can be envisaged, but unfortunately it would be very hard, if not impossible, to obtain computationally interesting self-replicating systems using "pure" cellular automata.

In fact, CA are, by definition, closed systems: all the information must be present in the array at iteration 0 (in our case, all the data for the system must be included in the initial loop). Since useful computation would require that each of the offspring execute a different function (or at the very least, the same function on different data), the requirement that all information be stored in the parent loop is too restrictive for our needs.

At this stage we therefore decided to stop further development of self-replicating machines in the cellular automaton environment, and attempt to transfer the accumulated experience to the design of our FPGA.

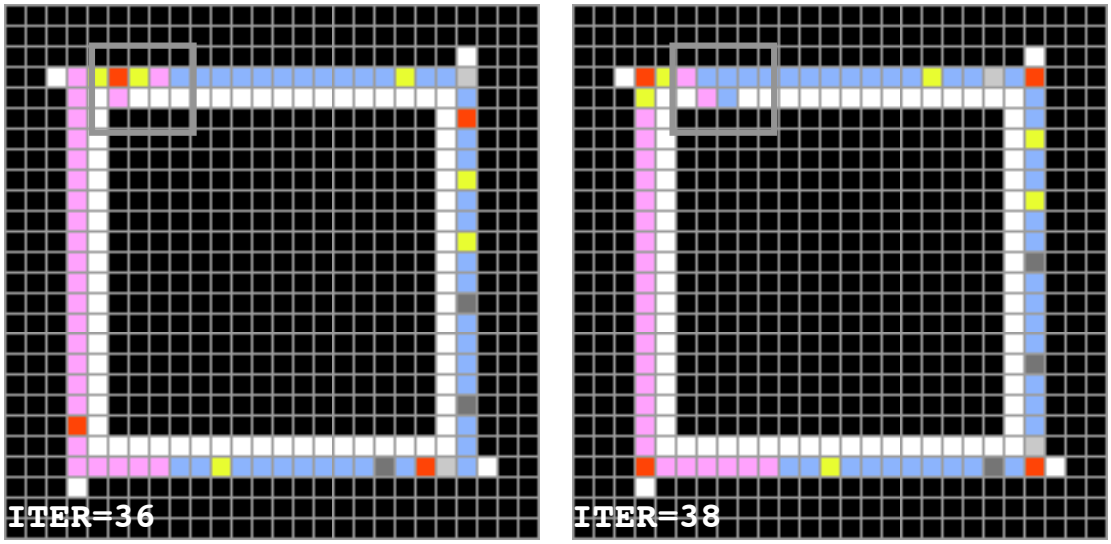

Figure 3-24: The initiation sequence opens a “door” in the sheath.

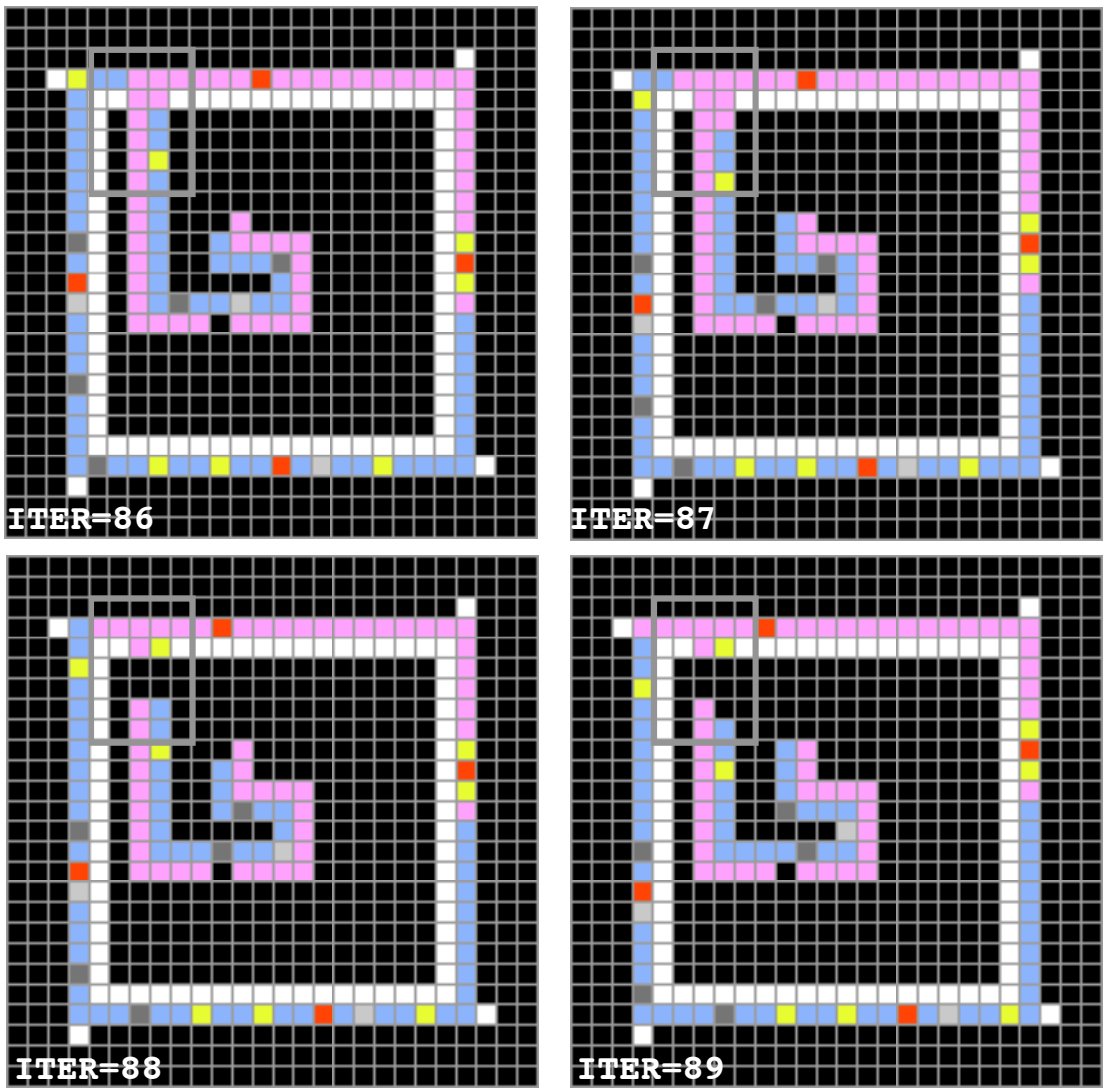

Figure 3-25: The copy of the program is concluded and the “door” is closed.

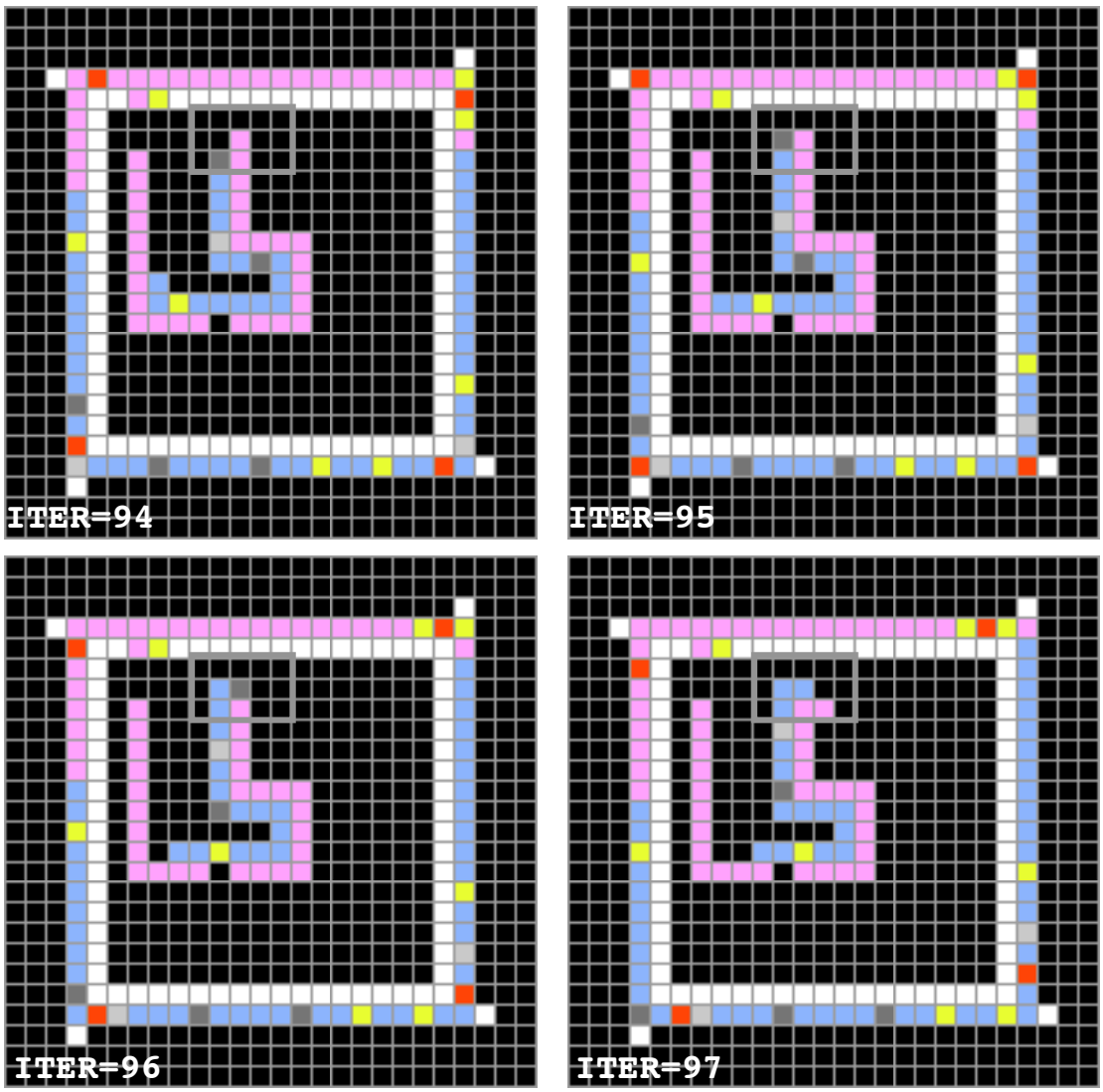

Figure 3-26: The execution of a “turn right” instruction.

3.5 Towards a Digital Hardware Implementation

An FPGA circuit is significantly different from a cellular automaton, a certain superficial resemblance notwithstanding. To develop a self-replication mechanism which can be efficiently adapted to digital electronic circuits in general and to FPGAs in particular, we therefore had to analyze the operation of our loop and attempt to extract not so much the precise mechanism used to achieve self-replication, but rather the general approach to the problem. In this section, we will present the results of this process, which led to the development of the *membrane builder*, a very simple cellular automaton which we then implemented in hardware and integrated into our FPGA.

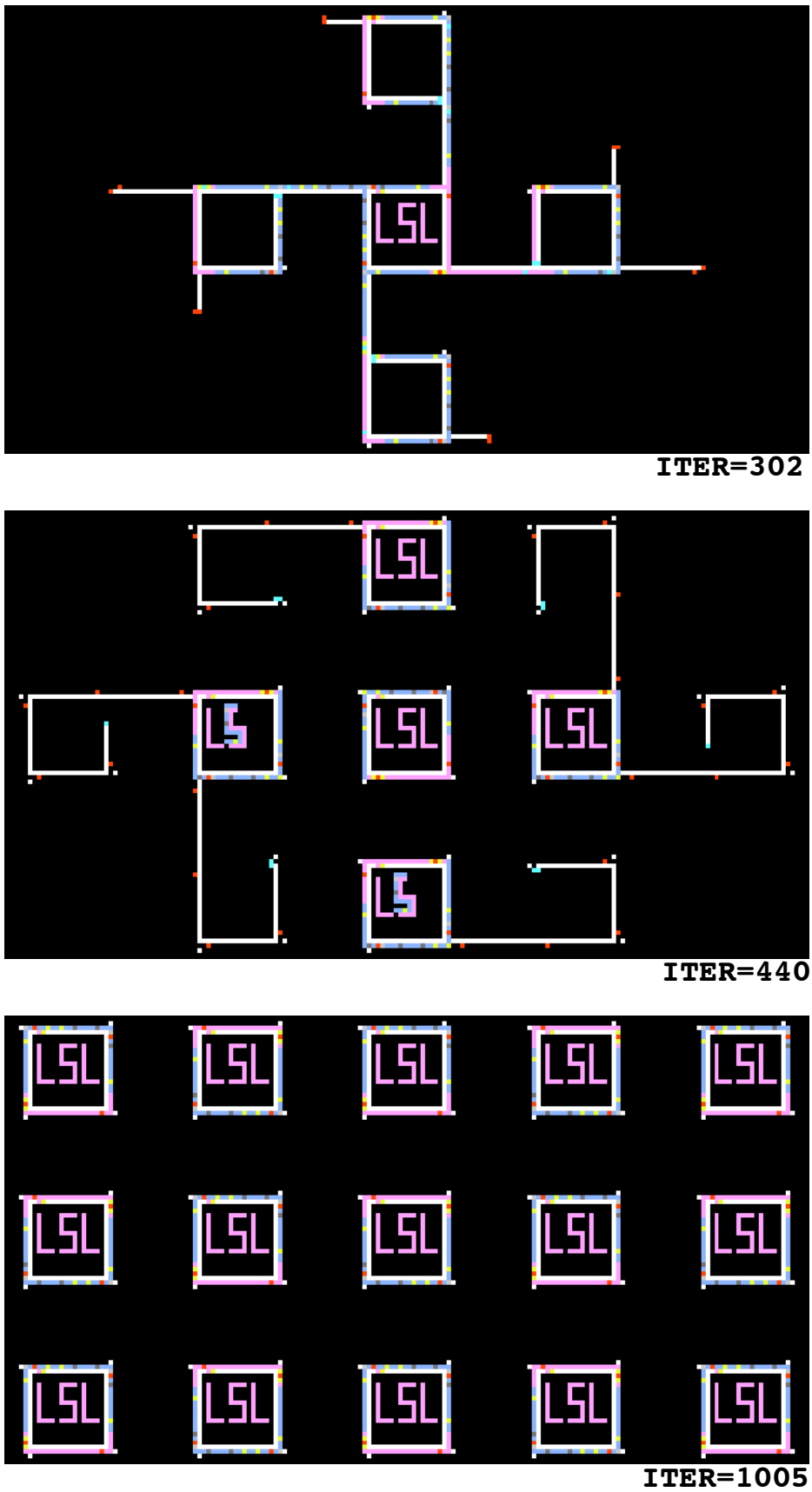

Figure 3-27: The program is copied and executed into each of the offspring.