Artificial Life. Vol. 4, No. 3, Summer 1998, pp. 259-282.

SELF-REPLICATING AND SELF-REPAIRING MULTICELLULAR AUTOMATA

Gianluca Tempesti†, Daniel Mange, André Stauffer
Logic Systems Laboratory
Swiss Federal Institute of Technology
CH-1015 Lausanne, Switzerland

Abstract

Biological organisms are among the most intricate structures known to man, exhibiting highly complex behavior through the massively parallel cooperation of numerous relatively simple elements, the cells. As the development of computing systems approaches levels of complexity such that their synthesis begins to push the limits of human intelligence, engineers are starting to seek inspiration in nature for the design of computing systems, both at the software and at the hardware levels. This paper will present one such endeavor, notably an attempt to draw inspiration from biology in the design of a novel digital circuit: a field-programmable gate array (FPGA). This reconfigurable logic circuit will be endowed with two features motivated and guided by the behavior of biological systems: *self-replication* and *self-repair*.

1 Introduction

Biological inspiration in the design of artificial machines is not a new concept: the idea of robots and mechanical automata as man-like artificial creatures predates even the development of the first computers. With the advent of electronics, the attempts to imitate biological systems in computing machines did not stop, even if their focus shifted from the mechanical world to the realm of information: since the physical substrate of electronic machines (i.e., the hardware) is not easily modifiable, biological inspiration was applied almost exclusively to information (i.e., the software).

Recent technological advances, in the form of programmable logic circuits, have engendered a re-evaluation of biological inspiration in the design of computer hardware. This paper introduces an attempt at exploiting this kind of inspiration in the design of digital circuits, and more particularly of one such programmable logic device, a field-programmable gate array (FPGA) known as MuxTree (for *tree of multiplexers*).

The next section will contain an overview of our overall approach to the design of bio-inspired hardware in general and of ontogenetic circuits (that is, circuits inspired by the biological processes involved in the development of a single organism, e.g., cellular division and cellular differentiation) in particular, introducing the Embryonics (for *embryonic electronics*) project. We will then describe in more detail the two bio-inspired features we implemented in our FPGA circuit, before concluding with a short analysis of our results.

† Corresponding author. Phone: +41-21-6932676. Fax: +41-21-6933705. E-mail: tempesti@di.epfl.ch.

2 Bio-Inspired Hardware and the Embryonics Project

The work presented in this paper is part of a more general research project, known as *Embryonics*, which aims at establishing a bridge between the world of biology and that of electronics, and in particular between biological organisms and digital circuits [19, 20, 33]. As the possible intersections between these two worlds are manifold, so the Embryonics project advances along more than one research axis, investigating domains as diverse as artificial neural networks [10] and evolutionary algorithms [21].

At the core of the project lies the *POE model* [27], which classifies bio-inspired systems along three axes: *phylogeny*, *ontogeny*, and *epigenesis*. Along the phylogenetic axis we find systems inspired by the processes underlying the evolution of species through time. Among the best-known examples of such systems are genetic and evolutionary algorithms [28]. Typical examples of epigenetic systems, inspired by those biological systems which are capable of learning (e.g., the nervous and endocrine systems), are artificial neural networks [23].

The phylogenetic and epigenetic axes of the POE model cover the majority of existing bio-inspired systems. The development of a multicellular biological organism, however, involves a set of processes which do not belong to either of these two axes. These processes correspond to the growth of the organism, that is, to the development of an organism from a single mother cell (the *zygote*) to a full-blown adult. The zygote divides, each offspring containing a copy of the genome (*cellular division*). This process continues (each new cell divides, creating new offspring, and so on), and each newly formed cell acquires a given functionality (i.e., liver cell, epidermal cell, etc.) depending on its surroundings, i.e., its position in relation to its neighbors (*cellular differentiation*). Ontogeny includes all these processes, which determine the development of an individual organism from the embryo to adulthood.

Cellular division is therefore a key mechanism in the growth of multicellular organisms, impressive examples of massively parallel systems: the $6x10^{13}$ cells of a human body, each a relatively simple element, work in parallel to accomplish extremely complex tasks (the most outstanding being, of course, intelligence). If we consider the difficulty of programming parallel computers (a difficulty that has led to a decline in their popularity), biological inspiration could provide some relevant insights on how to handle massive parallelism in silicon.

The work presented in this paper is concerned with the use of biologically-inspired mechanisms in the synthesis of digital circuits, and draws inspiration from two distinct sources. The first, as we have seen, is the biological mechanism of ontogeny: the complex behavior of natural organisms derives from the parallel operation of a multitude of simple elements, the cells. The second source of inspiration for our work is von Neumann's concept of self-replication of an universal computer, a mechanism which allows for the automatic creation of multiple identical copies of a machine from a single initial copy.

2.1 Von Neumann's Universal Constructor

The field of bio-inspired digital hardware was pioneered by John von Neumann. A gifted mathematician and one of the leading figures in the development of the field of computer engineering, von Neumann dedicated the final years of his life on what he called the *theory of automata* [36]. This research, interrupted by his untimely death in 1957, was inspired by the parallel between *artificial automata*, of which the paramount example are computers, and *natural automata* such as the nervous system, evolving organisms, etc.

Von Neumann conceived of a set of machines capable of many of the same feats as biological systems: evolution, learning, self-replication, healing, etc. His approach was based on the development of *self-replicating machines*, that is, automata capable of producing identical copies of themselves. Von Neumann identified a set of criteria to be met to obtain useful biological-like behavior in artificial machines. These criteria rested on two basic assumptions:

- Self-replication should be a special case of *construction universality*. That is, the self-replicating machines should be able not only to create copies of themselves, but also to construct any other machine, given its description.
- The self-replicating machines should be *universal computers*, that is, capable of executing any finite (but arbitrarily large) program. A class of automata capable of meeting this requirement was known to von Neumann: *universal Turing machines* [12].

Regrettably, the only machine von Neumann developed to any great extent was a theoretical model known as the *universal constructor* [5, 7, 25, 36], implemented using the mathematical framework of *cellular automata* (CA) [37]. Nevertheless, his theory of automata provides even today a firm foundation for the development of bio-inspired systems.

2.2 Field-Programmable Gate Arrays

Von Neumann's universal constructor was probably the first example of self-replicating computer hardware. Unfortunately, electronic technology in the fifties did not allow the development of so complex a machine. As a consequence, research on self-replicating hardware waned for several years. In the eighties, bio-inspiration gained new momentum under the label of *artificial life*, a research field pioneered by Christopher Langton which is attracting more and more interest in the scientific community. Under the impulse of new technology, bio-inspired hardware is also finally reaching the stage of physical realization [8, 11, 14, 19, 34].

The key technology that today allows the development of such approaches is the advent of programmable logic devices, usually referred to as *field-programmable gate arrays* (FPGAs) [4, 35]. These devices consist of two-dimensional arrays of identical elements, designed to be able to implement different functions, depending on the value of their *configuration*, a string of bits defined by the user at run-time. The size of an FPGA element (known as its *grain*) can vary considerably from one device to the next, ranging from complex look-up table based architectures (*coarse* grain) to much smaller hard-wired elements (*fine* grain). The elements are connected to each other through a connection network which is itself programmable.

A hardware designer can use an FPGA to implement just about any kind of digital logic circuit by defining the functionality of each element as well as the connections between these elements at run-time. Therefore, they are the ideal platform for the development of bio-inspired hardware, which requires that the layout of the circuit be modified through mechanisms such as self-replication, evolution, or healing (self-repair).

The goal of the work presented herein is to develop an FPGA architecture which exploits biologically-inspired mechanisms to introduce two novel features: self-replication and self-repair. The resulting circuit, a very fine-grained FPGA called *MuxTree* [19, 32, 33], was created with a specific application in mind: its use as a platform for the implementation of the complex bio-inspired structures we developed in the framework of the Embryonics project.

2.3 Ontogenetic Hardware

The two sources of inspiration we described (ontogeny and von Neumann's work) are different in very fundamental way. Both rely on a mechanism of self-replication to obtain arrays of elements which can be seen as processors, all executing an identical program. However, in von Neumann's case, the processors are universal Turing machines, and are identical in structure as well as in functionality: the phenomenon of cellular differentiation is entirely missing, and the whole system can be seen as a self-replicating *unicellular* organism. In nature, cells are different in structure and functionality (the appearance and behavior of a liver cell, for example, are considerably different from that of an epidermal cell), but any cell is potentially capable of replacing any other cell because it contains the description of the entire organism, i.e., the genome. Cellular differentiation is therefore at the very core of biological systems.

In Embryonics, we developed a solution that tries to integrate the two approaches: our system consists of an array of artificial cells implemented by small processors which have an identical structure (the same hardware layout) but different functionality (different software) [17, 18, 19, 33]. Our approach to creating ontogenetic hardware is thus based on the realization of computing systems (the artificial organisms) using an array of relatively small processing elements (the artificial cells), each executing the same program (the artificial genome). As we will see, this approach allows us not only to respect the basic definitions of a biological cell, but also to exploit some of the more specialized mechanisms on which the ontogenetic development of a cell is based.

2.3.1 The Artificial Organism

By demonstrating that it is possible to modify hardware using information, the development of FPGA circuits proved the feasibility of creating computer hardware inspired by biological ontogeny. To find a practical approach to the design of such systems, we turned to the essential features of biological organisms:

- In biology, an organism is a three-dimensional array of cells, all working in parallel to give rise to global processes (i.e., processes involving the entire organism). To respect the biological analogy, our artificial organism will be a two-dimensional array of elements working in parallel to achieve a global task, i.e., a given application.
- In biology, each cell contains the entire genome, that is, the function of every cell in the organism. To maintain the analogy between the genome and a computer program, we must regard the elements of our electronic organism as processors, each containing the same program. No single cell uses the entire genome, accessing only those portions needed for its function. Similarly, no single processor will execute all its program, but access a subset determined by its position within the array.

Drawing inspiration from biological organisms has thus led us to define our organism as a two-dimensional array of processing elements, all identical in structure (since each cell must be able to execute any subset of the genome) and each executing a different part of the same program. Such a system might not seem very efficient from the standpoint of conventional circuit design: storing a copy of the genome program in each processor is redundant, since each processor will only execute a subset. However, by accepting the weaknesses of bio-inspiration, we can also partake of its strengths. One of the most interesting features of biological organisms is their robustness, a consequence of the same redundancy which we find wasteful: since each cell contains a copy of the entire genome, it can theoretically replace any other. Thus, if one or more cells die as a consequence of a trauma (e.g., a wound), they can be recreated by any other cell. By analogy, if one or more of our processors should “die” (as a consequence, for example, of a hardware fault), they can theoretically be replaced by any other processor in the array. The redundancy introduced by the multiple copies of the genome thus provides an intrinsic support for self-repair, one of the main objectives of our research: by providing a set of spare cells (i.e., processors that are inactive during normal operation, but are identical to all others and contain the genome), we can (Fig. 1) reconfigure the array around faulty processors (of course, as in living beings, too many dead cells will cause the death of the organism).

Moreover, if the function of a cell depends on its coordinates, the task of self-replication is greatly simplified: by allowing our coordinates to cycle (Fig. 2) we can obtain multiple copies of an organism with a single copy of the program (provided, of course, that enough processors are available). Depending on the application and on the requirements of the user, this feature can be useful either by providing increased performance (multiple organisms processing different data in parallel) or by introducing an additional level of robustness (the outputs of multiple organisms processing the same data can be compared to detect errors).

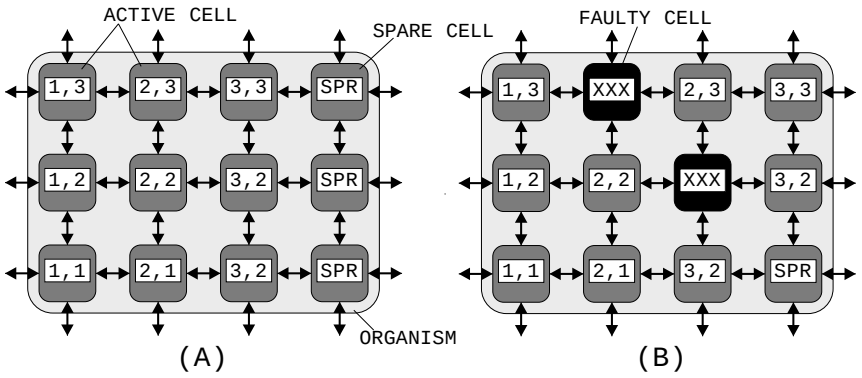

Figure 1: The electronic organism with no faults (A) and after reconfiguration (B).

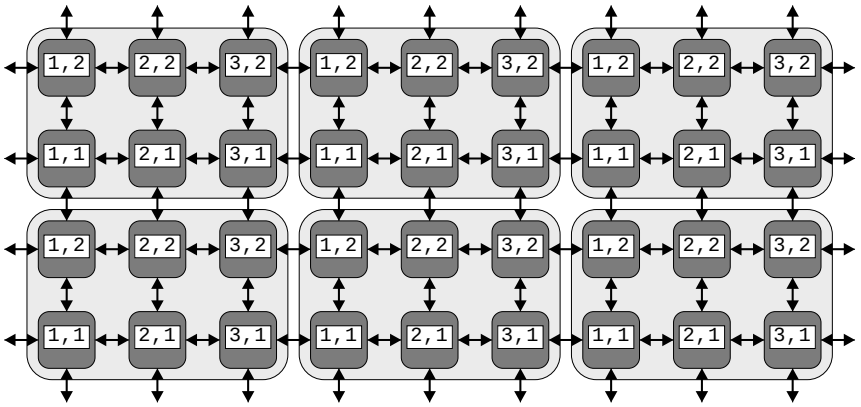

Figure 2: Multiple copies of the organism through coordinate cycling.

2.3.2 The Artificial Cell

Keeping in mind the requirements of the organism, we can now determine the basic features of our electronic cell. At the hardware level, all cells must be identical: since we want our organism to be reprogrammable, we cannot fix *a priori* the functionality of our cell. In addition, it has to be able to store the genome program with a coordinate-dependent access mechanism. The minimal features of our cells must therefore include (Fig. 3):

- A memory of configurable size to store the genome.
- An `[X,Y]` coordinate system, to allow the cell to find its position within the array, and thus its function.
- An interpreter to read and execute the genome.
- An application-dependent functional unit for data processing.
- A set of connections handled by a routing unit.

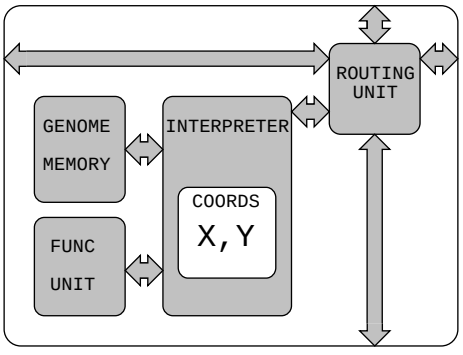

Figure 3: Structure of an electronic cell.

This cell is designed as a single element in a two-dimensional array of any finite size (the only limitation being the size of the coordinate registers). While the functional unit can theoretically be of any size and structure, biological organisms are a powerful example of systems whose complex behavior is derived not from the complexity of each component, but from the parallel operation of many simple elements. We aim to show that biological inspiration allows us, without excessive difficulty, to design complex systems based on very simple cells.

To demonstrate our cellular system, we designed a prototype, known as *MicTree* [17, 19], and used it to implement a set of applications which, while relatively simple because of the small number of available cells, are nevertheless interesting in that they exhibit both the properties of self-repair (if spare cells are available) and self-replication (if the array is large enough). The main limitation to their complexity is the fixed size of our cells: assigning *a priori* a maximum size for their components (e.g., its genome memory and coordinate registers) proved to be too restrictive to the versatility of the system. In order to obtain a truly universal computing machine, we need to tailor the components of the cell to the application.

Once again, biology provides a possible solution: the physical structure of a biological cell is determined by chemical processes occurring at the *molecular level*. After artificial organisms and artificial cells, we now need to define our artificial molecules. Fortunately, we are already familiar with a type of circuit capable of implementing our molecular layer: FPGAs. Using programmable logic as our molecular level allows us to maintain the analogy with biology: whereas a living cell consists of a three-dimensional array of molecules, a processor consists of a two-dimensional array of programmable logic elements. Furthermore, since in biology the most complex mechanisms of the cellular layer (notably, the genome) do not concern the molecular layer, the structure of our electronic molecule need not be unlike that of conventional FPGA logic elements. We are thus confronted with a three-layer system (Fig. 4), summarized in Table 1.

Biology	**Electronics**
Multicellular Organism	Parallel computer system
Cell	Processor
Molecule	FPGA element

Table 1: Analogies between biological and electronic systems in Embryonics.

2.3.3 The Artificial Molecule

In order to design an FPGA tailored for the requirements of the Embryonics project, we needed to introduce two biologically-inspired features: self-replication and self-repair.

The first feature is directly related to von Neumann's machine and is at the core of biological inspiration in Embryonics, since it allows the creation of arrays of artificial cells. In order to achieve a physical realization of von Neumann's machine, that is, the self-replication of a universal computer, our system requires the following capabilities:

- It should construct multiple copies of a machine from the description of a single specimen. Ideally, the machines themselves should generate and direct the process.
- The process should be applicable to machines capable of executing any given task.
- As a corollary, the process should be applicable to machines of any given size.

Our task in designing a self-replicating FPGA should therefore be obvious: since our artificial cells are indeed machines capable of executing any given task given a sufficiently large memory, in order to fulfill the requirements laid out by von Neumann our FPGA will require a mechanism capable of constructing multiple copies from the description of a single cell.

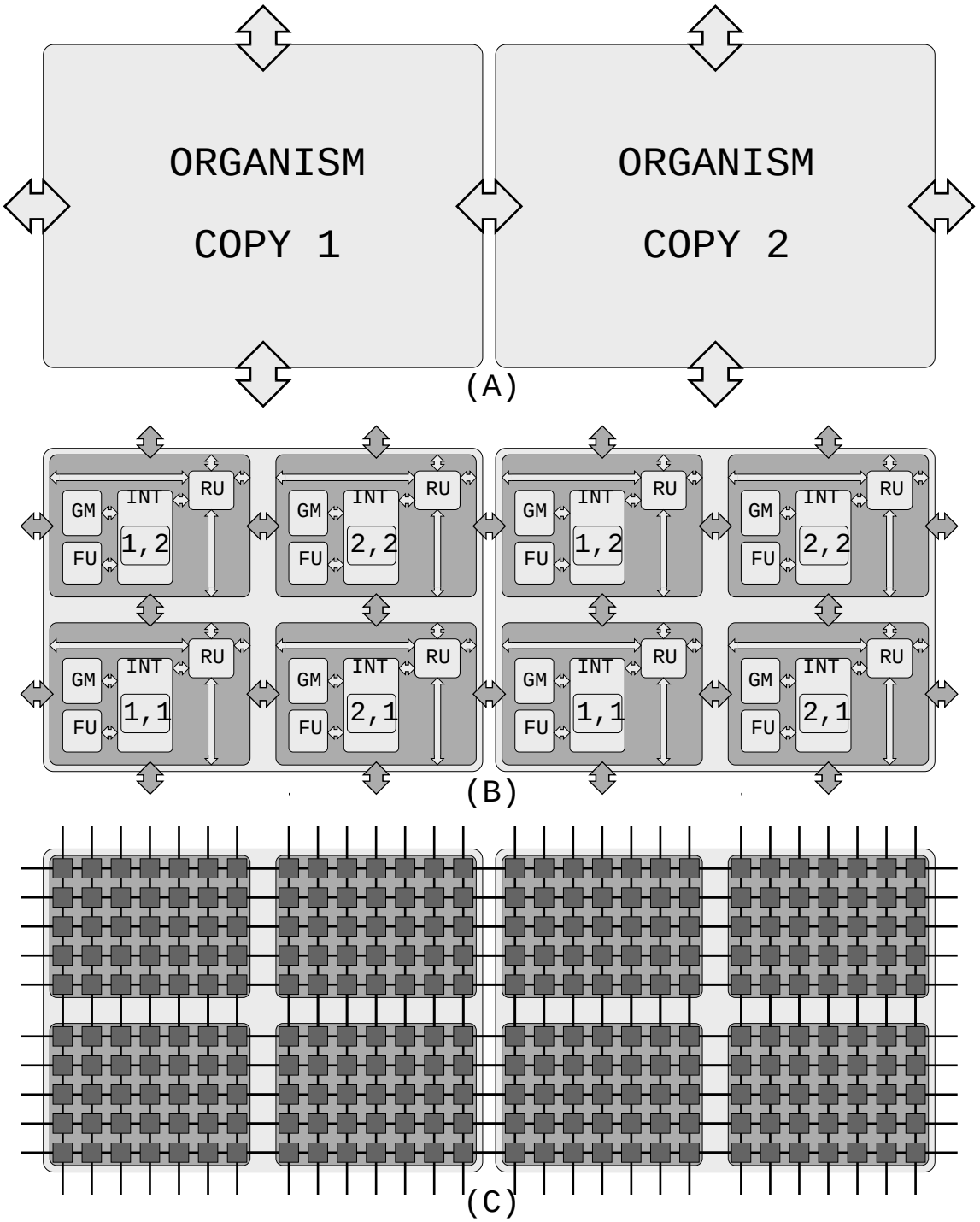

Figure 4: The three-level ontogenetic hardware: (A) organism (system) level, (B) cellular (processor) level, and (C) molecular (FPGA) level.

Self-repair, on the other hand, depends on a somewhat different set of assumptions, related more to engineering than to biology. In biology, as well as in von Neumann's approach, self-repair is achieved through the replacement of faulty *cells*. As we have seen, such a mechanism is present in our machines: faulty cells are replaced by identical spare cells whenever necessary. Introducing self-repair in our FPGA is the equivalent of cicatrization at the *molecular* level, a phenomenon which has no direct parallel in either biology or in von Neumann's work but is extremely interesting from an engineer's point of view for a number of reasons:

- A two-level self-repair system is likely to be more versatile and powerful than a single-level one. Self-repair at the molecular level implies that we will not need to sacrifice an entire cell (and thus a large amount of programmable logic) for every fault.
- Ideally, self-repair should be transparent to the user, occurring while the circuit is operating. Such a requirement is more likely to be fulfilled through dedicated hardware, and therefore at the molecular level.
- Self-repair mechanisms, and particularly the self-test mechanisms involved in fault detection, are very much dependent on the structure of the circuit being repaired. If most or all faults are detected at the molecular level, then achieving self-repair at the cellular level becomes much simpler.

From these observations, we can begin to outline the basic features of our ideal self-repair mechanism. First of all, it will require a transparent self-test mechanism capable of detecting as many faults as possible (ideally, *all* faults, but such a goal is not likely to be achievable). Moreover, such a system should be able to determine the exact location of the fault (that is, which of the elements is faulty) so that self-repair can restore the functionality of the array. As far as the self-repair mechanism itself is concerned, it should be able to repair as many faults as possible (again, it is unreasonable to assume that *all* faults will be repairable) and, in case of failure at the molecular level, activate the self-repair process at the cellular level.

Obviously, both these features will introduce additional hardware as well as additional delays in our circuit. Since our FPGA is, as we mentioned, extremely fine-grained, it will be very important to minimize the hardware overhead. On the other hand, the speed of operation is not an important factor in our research (biological systems, after all, operate relatively slowly). Our main effort in this project was therefore to minimize space (area) rather than time (delay), while respecting the considerable number of additional constraints imposed by the biological inspiration of our system.

3 Self-Replication

The design of a self-repairing and self-replicating FPGA presents a considerable challenge. While self-repair is a relatively well-investigated feature in the design of digital logic circuits, which implies an existing knowledge base for the development of our system, the same cannot be said for self-replication: research in this domain is relatively scarce, particularly where hardware is concerned. This lack compelled us to develop an entirely new self-replication mechanism, allowing us to arrive at an efficient hardware realization.

3.1 Langton's Loop

The complexity of Von Neumann's constructor derived from the attempt to implement self-replication as a particular case of construction universality. In 1984, Christopher Langton re-approached the problem from a different angle, by attempting to define the simplest cellular automaton capable exclusively of self-replication [16]. The automaton he developed, commonly known as *Langton's loop* (Fig. 5), is orders of magnitude simpler than von Neumann's.

Langton's loop derives its name from the dynamic storage of data inside a square sheath (the red elements 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 constantly circulates counterclockwise within the sheath, thus creating a loop.

The two instructions in Langton's loop are extremely simple: one tells the arm to advance by one position, while the other directs the arm to turn 90^o to the left. Obviously, after three such turns, the arm has looped back on itself, at which stage the connection between the parent and the offspring is severed, concluding the replication process. Once the copy is finished, the

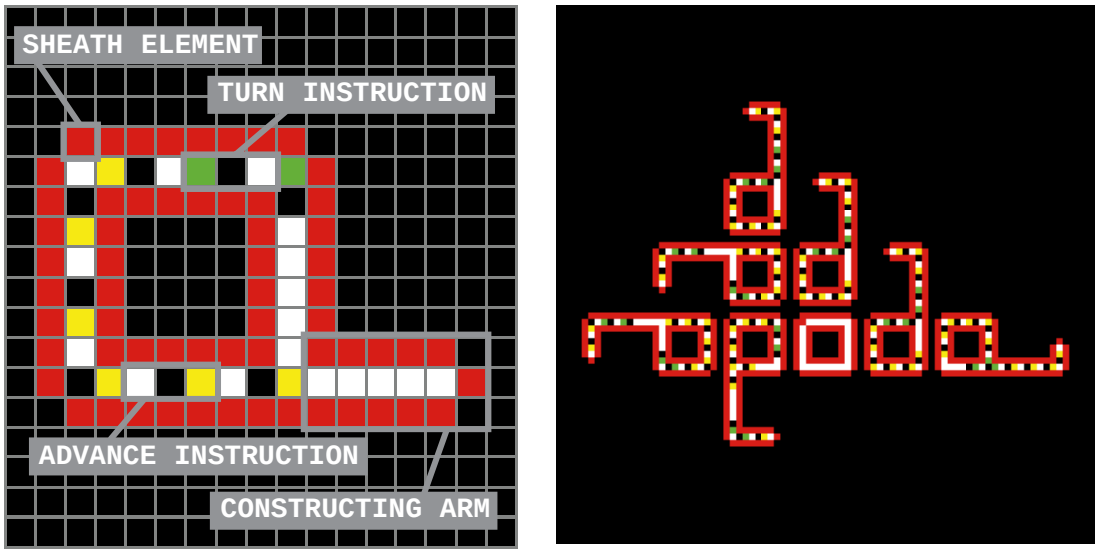

Figure 5: The initial configuration of Langton's loop (iter. 0) and its replication pattern (iter. 500).

parent loop proceeds to construct a second copy of itself in a different cardinal direction, while the offspring itself starts to replicate. The sequential nature of the self-replication process generates a spiraling pattern in the propagation of the loop through space: each loop tries to replicate in the four cardinal directions, until it finds the place already occupied either by its parent or by the offspring of another loop, and 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). Given its modest complexity, at least relative to von Neumann's automaton, Langton's loop has been thoroughly simulated.

Langton's loop 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.
2. It does not have any functionality beyond self-replication: the loop replicates and then dies. It is thus more similar to a biological (or a software) virus than to a cell.

These drawbacks notwithstanding, Langton's loop is the automaton that most closely approaches our requirements, and was the starting point of our research into self-replication.

To overcome Langton's loop's lack of functionality, we developed a relatively complex automaton (known as *Perrier's loop*) which exploits Langton's loop as a "carrier" for a two-tape universal Turing machine [24] (Fig. 6). The first operation of the automaton is let Langton's loop generate an offspring (iteration 158: note that the copy is limited to one dimension, since the second dimension is taken up by the Turing machine) whose main function is to determine a location for the copy of the Turing machine. Once the new loop is ready, a "messenger" runs back to the parent loop and start to duplicate the Turing machine, a process completely disjoint from the operation of the loop. When the copy is finished, 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).

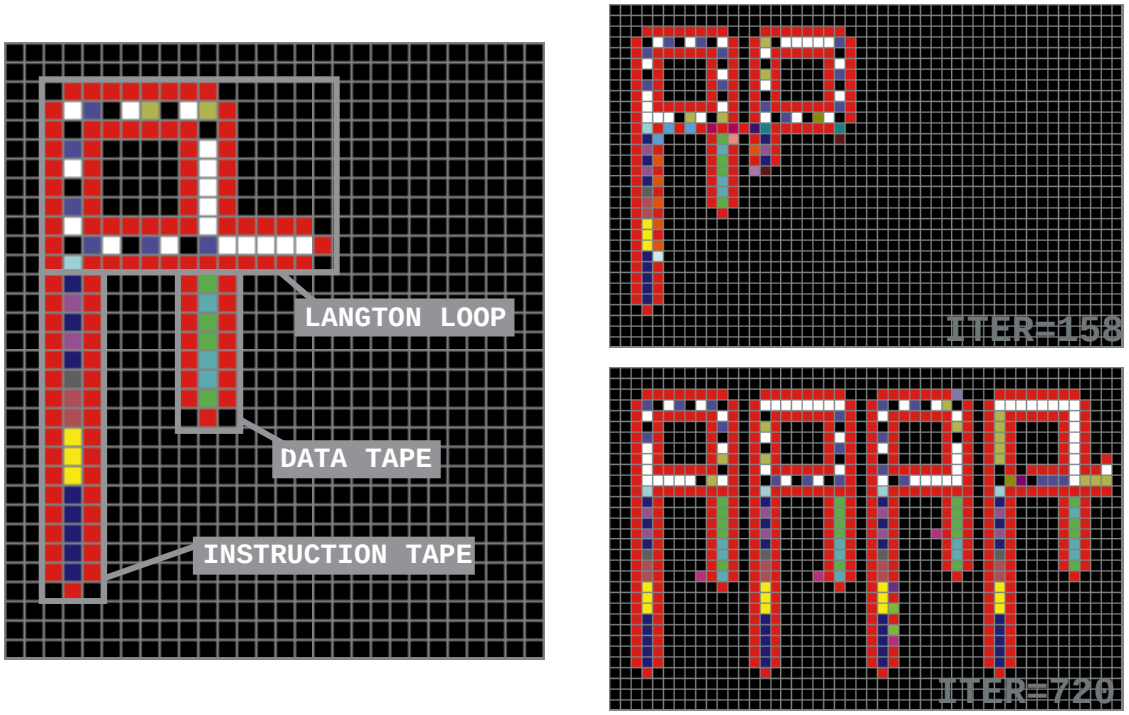

Figure 6: A two-tape Turing machine appended to Langton's loop (iter. 0) and its propagation.

The automaton thus becomes a self-replicating Turing machine, a powerful construct (as we mentioned, a universal Turing machine is capable of executing any finite program) which is unfortunately handicapped by its complexity, requiring 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 implementation. Adapting Langton's loop to fit our requirements thus proved too complex to be efficient, and we were forced to design a novel automaton to meet our requirements.

3.2 A Novel Self-Replicating Loop: Description

The self-replication mechanism of Langton's loop assumes that there is enough space for a copy of the loop, and the entire loop crashes upon meeting an obstacle (such as the border of a CA array). Modifying the automaton to overcome this drawback is very difficult: 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 automaton would need to be able to retract the constructing arm if it detects a boundary during the self-replication process. Such a mechanism would require major modification to Langton's loop: we thus decided to develop an entirely novel automaton.

In designing our self-replicating automaton [31, 33] (Fig. 7), we did maintain some of the more interesting features of Langton's loop. Notably, we preserved the structure based on a square loop to dynamically store information, as well as the concept of a constructing arm, in the tradition of von Neumann and his successors. However, in order to adapt the automaton to our requirements, we were forced to completely redesign the dynamics of the self-replication mechanism. To mention but a few of the more distinctive modifications:

- We extend four constructing arms in the four cardinal directions at the same time. When the arm meets an obstacle (e.g., the border of the array), it retracts, allowing our automaton to operate in a finite space.

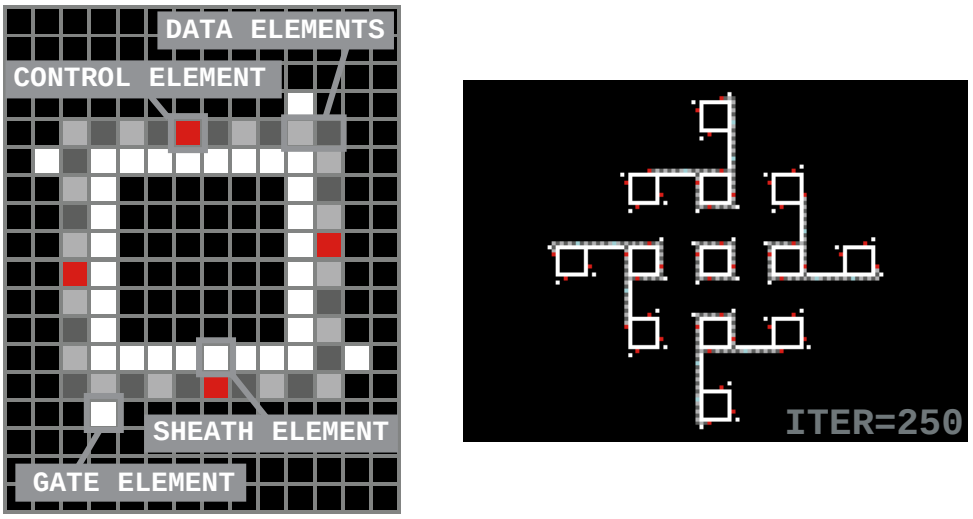

Figure 7: Our novel self-replicating loop and its propagation pattern.

- The arm does not immediately construct the entire loop. It begins by building a sheath of the same size as the original, and only then the data circulating in the loop is duplicated and the copy is sent out to wrap around the new sheath. As we will see, dividing the self-replication process in two phases is an important advantage for the transition to digital hardware.
- As a consequence of the above, rather than using all of the data in the loop to direct the constructing arm, we need only four *control elements*. The remaining *data elements* can be used to implement other functions, allowing us to overcome the second drawback of Langton's loop (its lack of functionality apart from self-replication).

As should be obvious from the above, while our loop owes to von Neumann the concept of a constructing arm and to Langton the basic loop structure, it is in fact a very different automaton, endowed with some of the properties of both. The complexity (i.e., the number of transition rules) of the basic configuration of our loop is of the same order of magnitude as that of Langton's loop, with the proviso that it is likely to increase drastically if the data in the loop is used to implement a function.

3.3 A Novel Self-Replicating Loop: a Functional Example

In Fig. 8, we illustrate an example of how the data states can be used to carry out operations alongside self-replication. In this case, the operation 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 used five data states, each an instruction for the construction of the letters: *advance*, *turn left*, *turn right*, *empty space*, and a NOP (no operation) instruction. This program requires 330 additional rules and 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. As it loops, the program is duplicated and a copy sent through the door to the interior of the loop, where it constructs the letters. At the end of the process, the door is closed.

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" instructions cause the arm to change direction, and the "empty space" instruction produces a gap in the arm (so as to separate the letters).

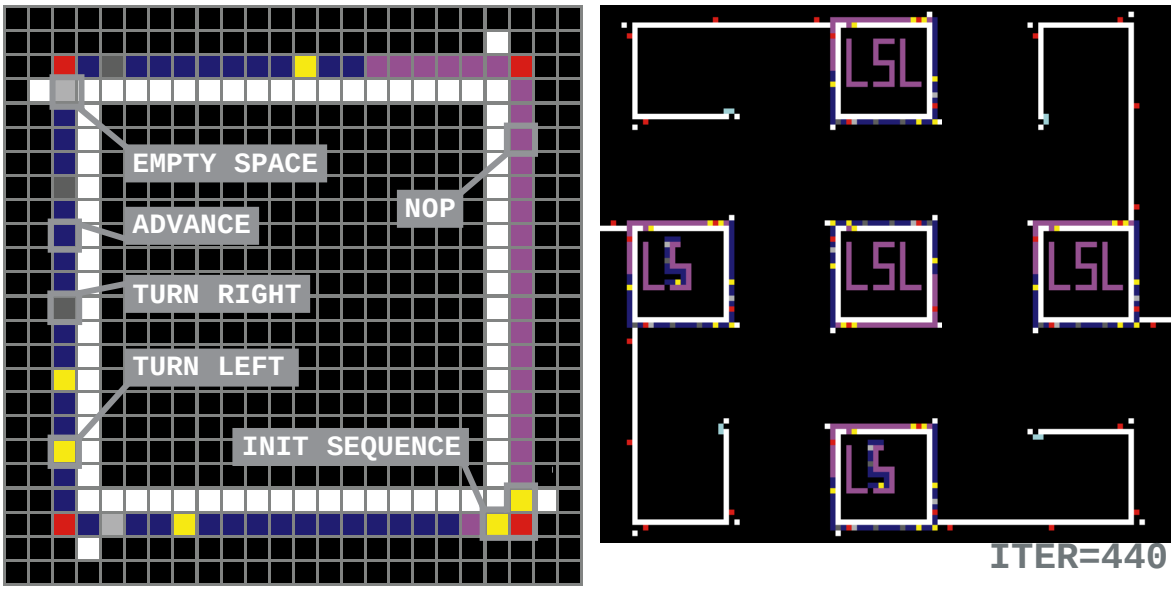

Figure 8: Configuration of the LSL automaton at iteration 0 and its propagation.

During the self-replication of the loop, 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.

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, CAs 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.

Therefore, at this stage we 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.

3.4 The Membrane Builder

The self-replicating loop we developed corresponds for the most part to our requirements: it is a computing machine capable of self-replication which can exist in a finite space and execute non-trivial functions. The transition from cellular automata to hardware, however, requires a process of synthesis: cellular automata are very inefficient from the point of view of an hardware realization, notably because every element needs to access a lookup table of considerable size. In order to find a viable approach to the realization of self-replicating hardware, we therefore had to try to extract from our cellular automaton the essential features of the self-replication mechanism, to use them as a basis for the design of an electronic circuit.

One of the main differences between our loop and Langton's lies in the two-phase mechanism of self-replication. In fact, while in Langton's loop the process is indivisible, we can identify two distinct phases in the self-replication of our loop: a *structural phase*, where the "skeleton" of the offspring is set into the empty space, and a *configuration phase*, where the functionality of the parent is copied into the offspring.

While the configuration phase, for a number of practical reasons, is not suited to an FPGA implementation, the structural phase can indeed be adapted to hardware. In particular, if consider an FPGA before configuration as an array of CA elements in the quiescent state, we can

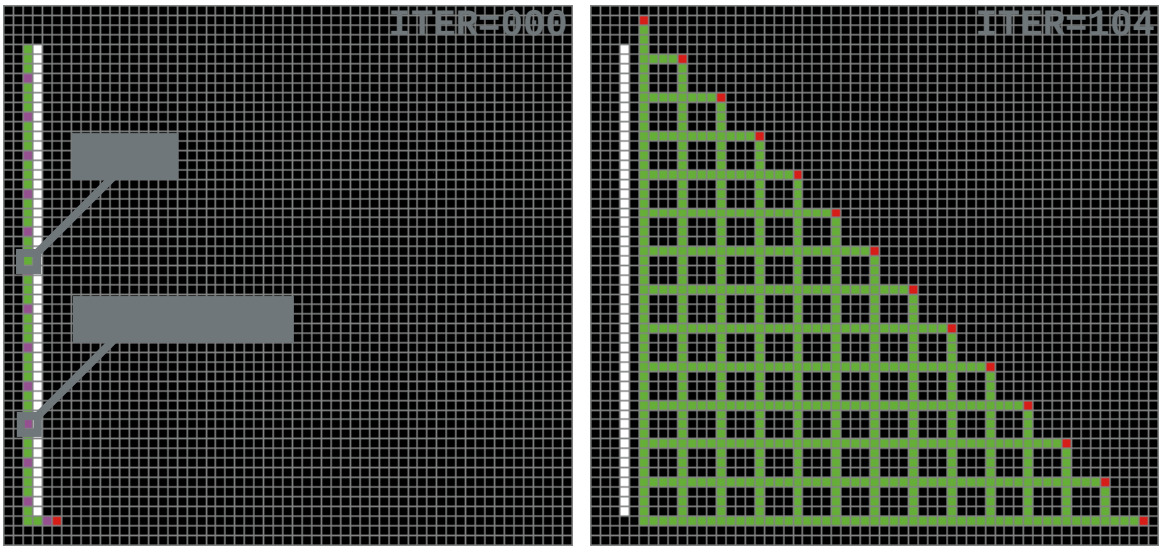

Figure 9: The membrane builder at iteration 0 and after the end of propagation.

think of the structural phase of self-replication as a mechanism which *partitions* the FPGA into identical *blocks* of molecules of programmable size (each block will then contain a single artificial cell), a task which can be realized by an extremely simple cellular automaton (Fig. 9).

The automaton, simple enough to allow a trivial implementation in hardware, can transform (through a simple process which we will not describe in detail) a one-dimensional string of states (analogous to a configuration bitstream residing in a memory chip) to a two-dimensional structure.

In order to integrate the automaton to our FPGA, we inserted the CA elements in the spaces between the FPGA elements (Fig. 10). By entering the appropriate sequence of states, we can then partition the array into identical blocks of variable size, creating a *cellular membrane* which we can use to direct the propagation of the FPGA's configuration (Fig. 11).

By exploiting the experience accumulated in designing our self-replicating loops, we were thus able to create a very simple self-replication mechanism for our FPGA. At this stage, we turned our attention to the second bio-inspired feature of our FPGA: self-repair.

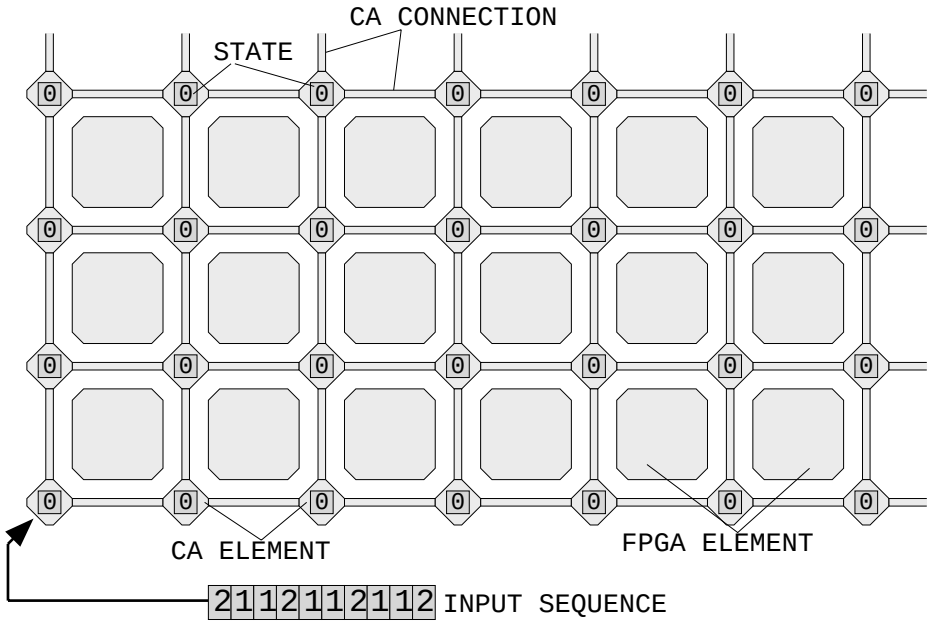

Figure 10: Definition of the membrane using a simple cellular automaton.

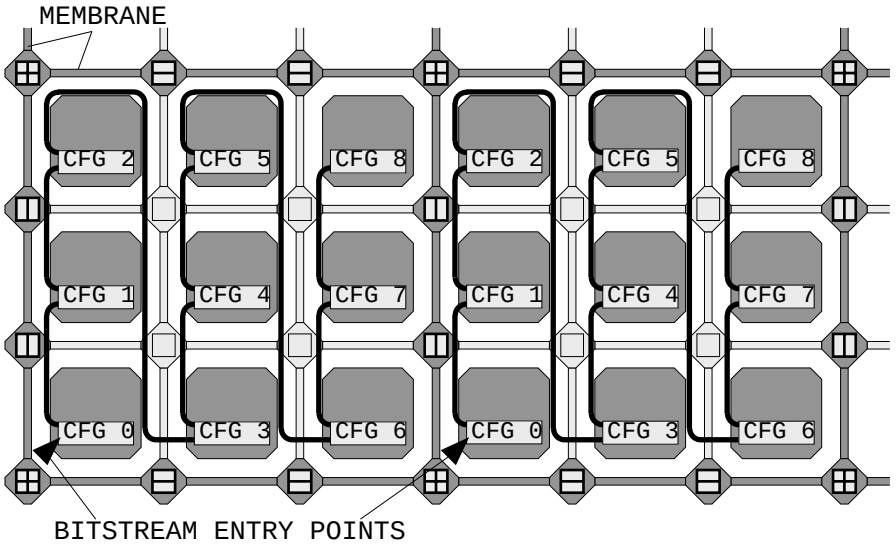

Figure 11:

4 MuxTree and Self-Repair

The architecture of an FPGA element can vary considerably from one circuit to the next. The only actual requirement is that it must be possible to implement any given function using one or more elements. In addition, it is customary, if not strictly required, to include some form of memory in an element so as to be able to easily implement sequential systems (that is, systems which contain sequential elements such as flip-flops, registers, etc.)

MuxTree [19, 32, 33], the FPGA we developed for the Embryonics project, is no exception to this rule, but is unusual in that it is remarkably fine-grained: like all FPGAs, it is a two-dimensional array of elements (the molecules of our three-level system) which, in MuxTree's case, are particularly small. Each element, in fact, is capable of implementing the universal function of a single variable and of storing a single bit of information.

The basic element of our FPGA (Fig. 12) can be seen as composed of three separate subsystems: the programmable function (FU), the programmable connections (SB), and the configuration register (CREG).

The programmable function is realized using a single two-input multiplexer. The multiplexer being a universal gate (i.e., it is possible to realize any function given a sufficient number of multiplexers), the first requirement for an FPGA element is respected. In addition to the multiplexer, each element is also capable of storing a single bit of information in a D-type flip-flop, thus fulfilling the second requirement.

As far as the programmable connection network is concerned, a MuxTree element contains two separate sets of connections: a fixed short-distance network for communication between immediate neighbors, and a programmable long-distance network for distant elements. The latter is controlled by a switch box (SB) which can route the output NOUT of an element to its four neighbors, as well as propagate signals in the four cardinal directions.

The element's function and connections are determined by a 17-bit configuration string, stored in the shift register CREG. These bits are sufficient to configure both the programmable function and the connection networks.

MuxTree's most remarkable feature is probably the unusual structure of its connections (and particularly of its fixed short-distance network), designed to allow an array of MuxTree elements to be easily configured as a *binary decision diagram* (BDD) [3]. Since BDDs are well-known representation methods for logic functions, this structure becomes very useful when calculating the configuration required to implement a given function using MuxTree.

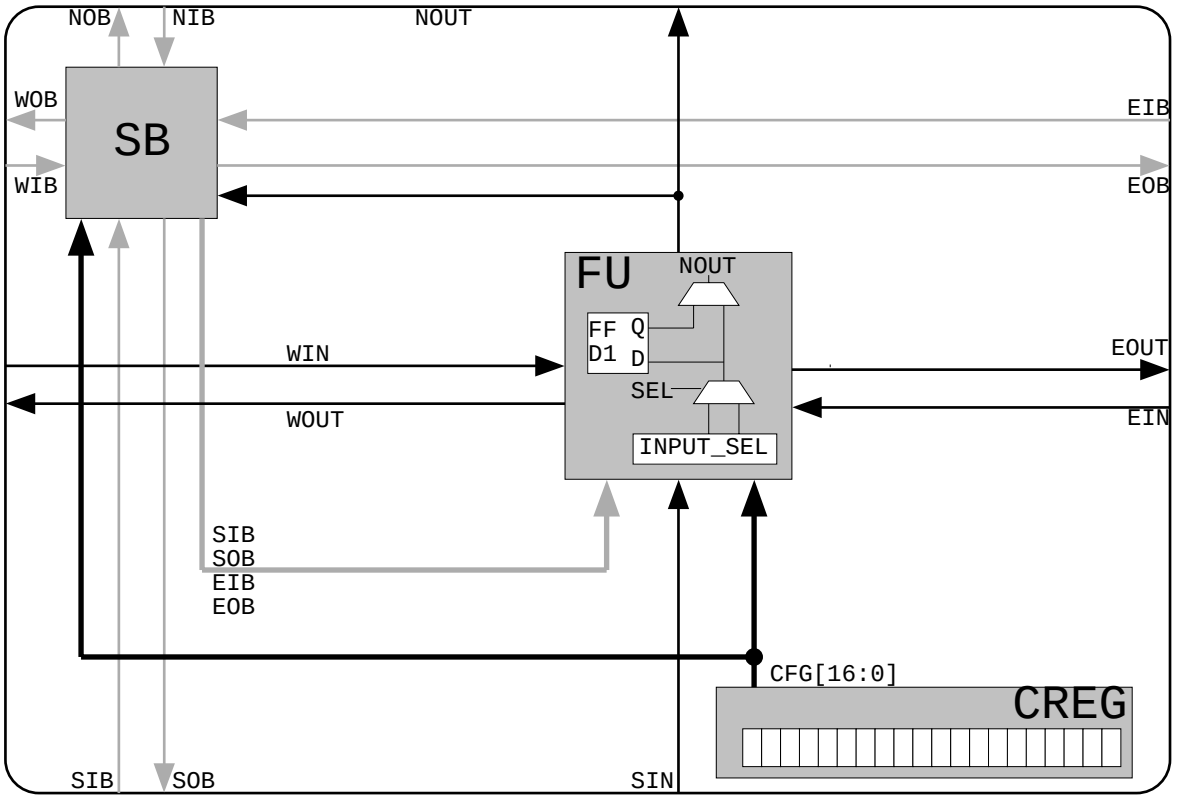

Figure 12: Overall structure of a MuxTree element.

4.1 Self-Test in MuxTree

Any literature search, however superficial, on the subject of testing will reveal the existence of a considerable variety of approaches to implementing self-test in digital circuits [1], including some which can be applied to FPGAs [2, 13, 29]. Even though we exploited to the greatest possible extent this existing knowledge base in our system, we found that the special requirements of our bio-inspired systems prevented the use of off-the-shelf approaches.

The first non-conventional constraint was imposed by the need of the self-repair mechanism to know not only that a fault has occurred somewhere in the circuit, but its exact site. Thus, our system has to be able to perform not only *fault detection*, but also *fault location.*

A second constraint is that we desired our system to be completely distributed, preventing the use of a centralized control, very common in conventional self-test systems. This constraint is due to our approach to reconfiguration: by assuming that any element can be replaced by any other (an assumption which, as we will see, is a direct consequence of our self-replication mechanism), we need our array to be completely homogeneous.

The relatively small size of the MuxTree elements also imposed a constraint on the amount of logic allowed for self-testing. While the minimization of the logic was not our main goal, we nevertheless tried to keep the testing logic down to a reasonable size, which imposed some rather drastic limitations on the choices available for the design of our mechanism.

In addition to these “imposed” constraints, we also decided to attempt to design a self-test mechanism capable of operating on-line, that is, while the system is executing the application and transparently to the user. This self-imposed constraint is extremely strong: to obtain an on-line self-test system while at the same time minimizing the amount of additional logic is an extremely arduous task. As a consequence, in our actual implementation this constraint was somewhat relaxed, and we will see that our self-test will occur on-line only partially.

Once again, we will exploit the observation that the MuxTree element can be divided into three subcircuits to separately analyze possible self-test mechanisms for each subcircuit.

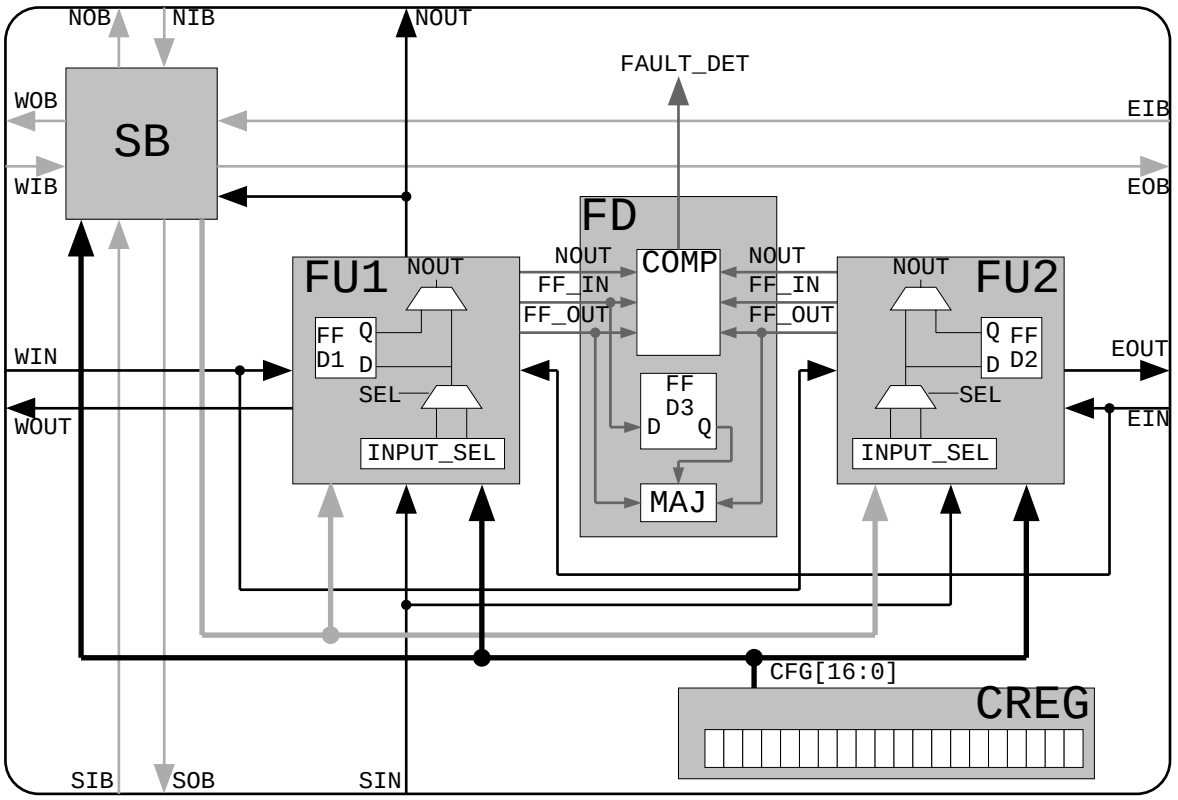

Figure 13: A third flip-flop and a 2-out-of-3 majority function allow us to recover the correct value even if a fault has damaged one of the flip-flops.

4.1.1 The Programmable Function

Thanks to the experience we acquired in the design of an integrated circuit containing an early prototype of MuxTree, we were able to determine that the functional part occupies approximately 10% of the total silicon area of an element. Considering its small size, we decided to test the function through duplication. Obviously, the presence of two identical copies of the sub-circuit allows the detection of faults through a comparison of their outputs.

Such a comparison would be sufficient to assure fault detection. However, our task is complicated by the need for self-repair: to allow the circuit to resume operation after being repaired, we must be able to preserve its *state*, i.e., the information stored in its memory elements. This requirement forced us to introduce additional logic so as to be able to recover the value stored in the flip-flops should one of them prove faulty.

To this end, we had to add to our self-test logic (Fig. 13) an additional comparison between the *inputs* of the two flip-flops, so as to prevent an incorrect value from being stored, as well as a third copy of the flip-flop, so as to recover the correct memorized value with a simple 2-out-of-3 majority function.

4.1.2 The Programmable Connections

MuxTree is a relatively connection-intensive circuit. In fact, while the network joining the elements is not very complex, the small size of the functional part implies that a fault is at least as likely to occur in a connection as in the few logic gates which implement the element's functionality. Solutions to the problem of testing connections in an FPGA do indeed exist, but invariably require a considerable amount of redundancy through duplication. While not excluding the possibility of introducing it in the future, we deemed that the advantages to be gained from the test of the connections did not justify the considerable hardware overhead, at least in our current implementation.

4.1.3 The Configuration Register

Testing the configuration register poses similar problems, but its considerable size (about 80% of the surface of an element) makes testing imperative. As for the rest of the element, the register's self-test mechanism must require a limited amount of additional logic, should ideally be on-line and transparent, and should be compatible with self-repair. It is this last requirement which proved most restrictive: reconfiguring the array implies that a spare element will assume the function of the faulty one, and thus its configuration. Since each element contains a single copy of the configuration register (its size precludes duplication), a fault which modifies the value stored in the register results in an unrecoverable loss of information.

So again we were not able to meet all our requirements. However, the probability of a fault occurring in the register being greater than for any other part of the element, we decided that some degree of testing was indispensable. We determined that relaxing the requirement that the self-test occur on-line would allow us to design an extremely simple mechanism.

Our approach is based on the observation that, when the circuit is programmed, the configuration bitstream is shifted into the register. It is therefore simple to add to the bitstream a *header*, in the form of a dedicated testing configuration sent *before* the actual configuration (so as to avoid any loss of information). This header, shifted into all the registers in parallel, will consist of a pattern designed to exhaustively test that the configuration register is operating correctly. We will not describe the pattern in detail: suffice it to say that such a pattern (indeed more than one) does exist, and is capable of detecting any fault in the register.

In conclusion, by relaxing the requirement that the detection occur on-line, we were able to design an extremely simple fault detection system (the hardware overhead consists of only a few logic gates) which, as we will see, is perfectly compatible with self-repair.

4.2 Self-Repair in MuxTree

As was the case for self-test, there exist a number of well-known approaches to implementing self-repair in two-dimensional arrays of identical elements [9, 15, 22]. Most, if not all, rely on two main mechanisms: *redundancy* and *reconfiguration*. The system we developed to implement self-repair in MuxTree is no exception, even if it had to satisfy a set of relatively non-standard constraints imposed by the unique features of our FPGA.

A mechanism which allows an electronic circuit to be repaired need obviously be very different from that exploited by nature in biological organisms. Since physically repairing a hardware fault is impossible, we must provide a set of spare elements (redundancy) and a mechanism to let them replace faulty elements in the array, that is, we need a mechanism to reroute the connections between the elements (reconfiguration).

As was the case for self-test, the small size of our elements imposes major limitations to the amount of logic we can introduce to implement a self-repair mechanism. This limitation has serious implications for the choice of possible reconfiguration schemes.

Unlike self-test, our self-repair process can occur off-line, that is, cause a temporary interruption of the operation of the circuit (of course, the process should be as fast as possible). However, it is fundamental that the *state* of the circuit (that is, the contents of all its memory elements) be preserved through the self-repair process, so as to allow normal operation to resume after the reconfiguration is complete.

In practice, the only memory elements in our system are the configuration register and the flip-flop inside the functional part of the element. Since our self-test system allows us to test the register only during configuration, we can also limit its repair to the configuration phase. The assumption that the register is fault-free during normal operation is extremely useful for the reconfiguration of the array, and effectively reduces the requirement that the state of the circuit be preserved to the need to prevent the loss of the value stored in the flip-flops.

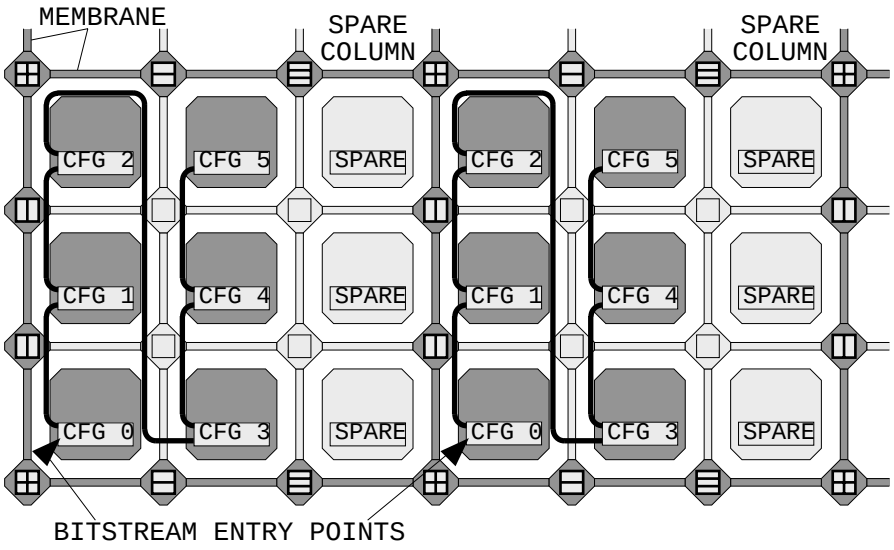

Figure 14: Self-replication of the configuration bitstream in the presence of spare columns.

4.2.1 Self-Replication Revisited

At this stage, we turned our attention back to the self-replication mechanism. A desirable, if not strictly necessary, constraint on our self-repair system would be that the reconfiguration be contained within each single block, and thus within each cell. In other words, the spare elements should be part of each block and contained within the cellular membrane. This requirement is far from trivial, since it implies that the location of the spare elements be a function of the size of the block, which is programmable and thus unknown *a priori*. Fortunately, by exploiting existing hardware, and notably the membrane-building CA, we can meet this requirement with a remarkably small amount of additional logic.

It is in fact fairly simple to modify the automaton so as to use the membrane itself to define which of the columns of the array will contain spare elements (Fig. 14). Simply by adding one additional state to the automaton, we obtain a very powerful system: this approach allows us not only to limit reconfiguration to the interior of a block, but also to program the robustness of the system. In fact, by adding or removing these special states to or from the CA input sequence, we are able to modify the frequency of spare columns, and thus the capability for self-repair of the system. Without altering the configuration bitstream of the MuxTree elements, we can introduce varying degrees of robustness, from zero fault tolerance (no spare columns) to 100% redundancy (one spare column for each active column).

4.2.2 The Reconfiguration Mechanism

In order to take advantage of the spare elements, we require a mechanism to transfer the information stored in a faulty element (notably, its configuration plus the value stored in its flip-flops) to one of the spare elements.

Our mechanism for repairing faults relies on the reconfiguration of the network through the replacement of the faulty element by its right-hand neighbor: the configuration of the faulty element, together with the value stored in its flip-flop, are shifted into the neighbor. The configuration of the neighbor will itself be shifted to the right, and so on until a spare element is reached (Fig. 15). Once the shift is completed, the faulty element “dies” with respect to the network: the connections are rerouted to avoid it, an operation which can be effected very simply by diverting the north-south connections to the right and by rendering the element transparent to the east-west connections. The array, thus reconfigured and rerouted, can then resume executing the application from the same state it held when the fault was detected.

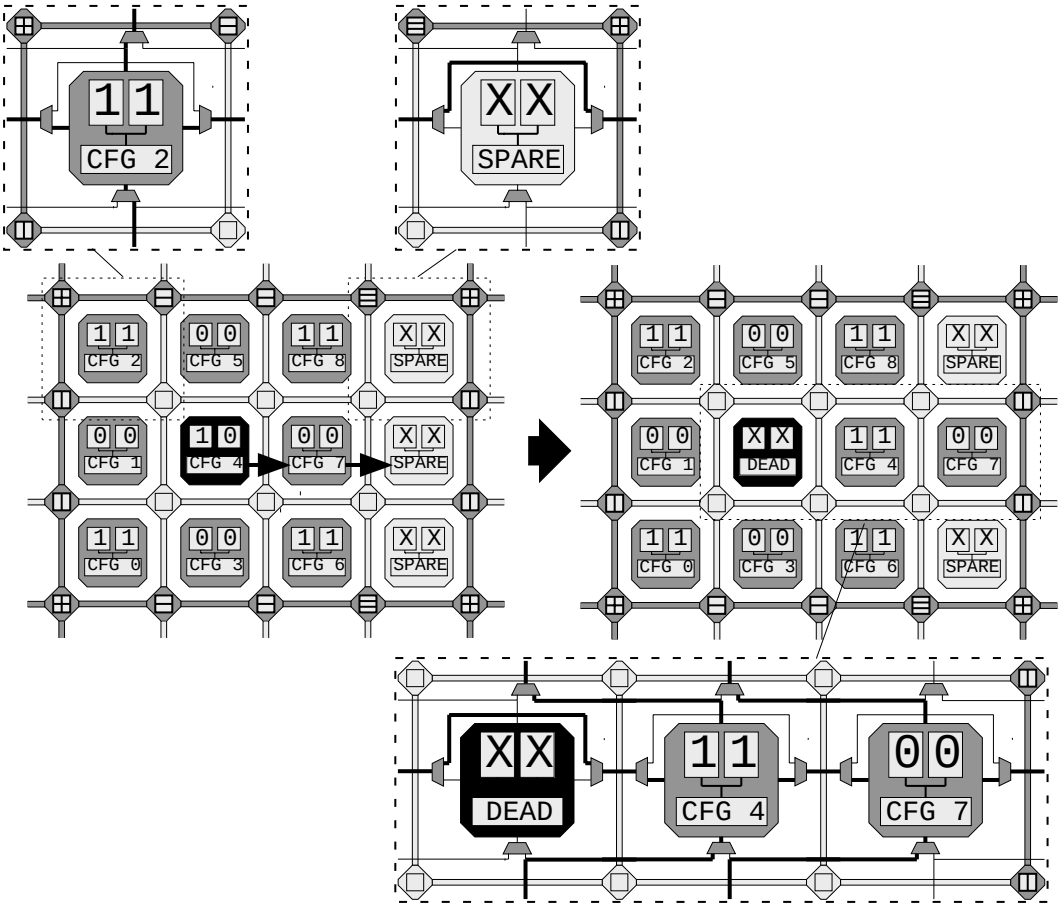

Figure 15: A fault is detected and the array is reconfigured to avoid the faulty element.

When a fault is detected, the FPGA goes off-line for the time required by the reconfiguration (somewhat like an organism becoming incapacitated during an illness), an interruption which can fortunately be minimized: the reconfiguration, being an entirely local mechanism, can exploit a faster clock signal, thus limiting the duration of the process.

Rerouting the connections around a faulty element requires a set of multiplexers to select alternate connecting paths. To minimize the hardware overhead, we limit the reconfiguration to a single column, that is, we do not allow the configuration of an element to be shifted more than once, restricting the number of repairable faults to one per row between two spare columns. This limitation, of course, is partially compensated by the programmability of the frequency of the spare columns,

4.2.3 MuxTree and MicTree

However versatile MuxTree's self-repair system might be, it is still subject to failure, either because of saturation (if all spare elements are exhausted) or because a non-repairable fault is detected. Should such a failure occur, we need to activate the self-repair mechanism at the cellular level (see above in section 2.3.1). To this end, we designed a `KILL` signal which is propagated through an entire column of blocks (Fig. 16). Since a block is ultimately meant to contain one of our artificial cells, killing a column of blocks is equivalent to deactivating a column of cells. At the cellular level, this event will trigger a recomputation of the coordinates of all cells in the system, that is, will activate the cellular-level reconfiguration mechanism (Fig. 17). In other words, the robustness of the system is not based on a single self-repair mechanism, which might fail under extreme conditions, but rather on two separate mechanisms which cooperate to prevent a fault from causing a catastrophic failure of the entire system.

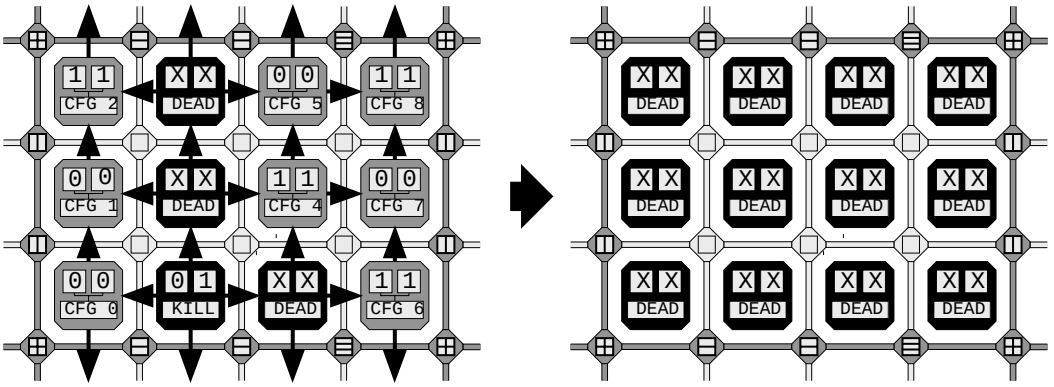

Figure 16: The saturation of the molecular-level repair mechanism causes a column of blocks to be killed.

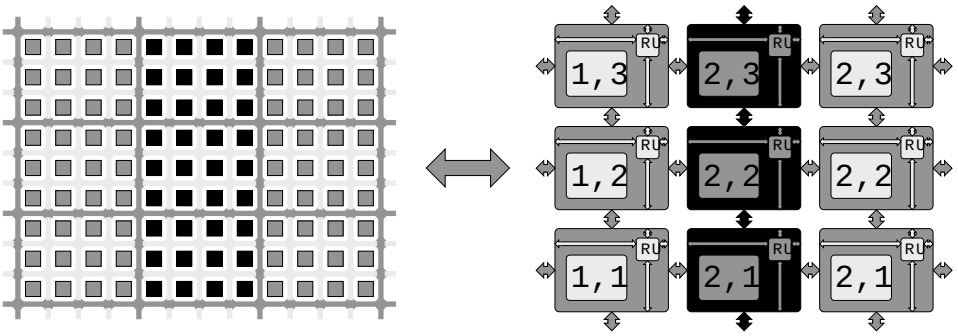

Figure 17: The two-level self-repair system.

5 Conclusion

Our goal in this paper was to demonstrate the feasibility of designing an FPGA capable of self-replication and self-repair. As far as self-replication is concerned, our goal was met in most respects: our mechanism is capable of generating multiple copies of our artificial cells from the description of a single specimen, of any given size and thus capable of executing any given task. The only compromise is the use of an external source for the configuration of the cells (ideally it should be the cells themselves which generate and control the replication process). In our case, the replication process is indeed controlled by the dedicated hardware integrated in our FPGA, but the configuration bitstream is generated outside the circuit itself and not by the cells. The achievement of such an "ideal" system remains a future research goal.

For self-repair, the results are not quite as close to optimal, notably where the self-test mechanism is concerned. The constraints of biological inspiration, coupled with the need to minimize the hardware overhead, proved too strong to allow on-line self-test of more than a relatively small part of the circuit. However, our system can detect faults in most of the circuit (an accurate estimate of the fault coverage is impractical at this stage, as MuxTree is still far from its final implementation) through off-line self-test during configuration.

On the other hand, the self-repair mechanism itself fully meets our requirements: it can repair of a considerable number of faults (the coverage depends, of course, on the number of spares assigned by the user), it is capable of activating self-repair at the cellular level through its global `KILL` signal, and, if not always transparent, it is remarkably fast. In addition, our system exceeds our requirements with the introduction of programmable redundancy, which allows the user to determine the amount of logic to be "sacrificed" for additional coverage.

In conclusion, we designed an FPGA capable of realizing with ease complex systems which, inspired by the ontogenetic processes of cellular division and cellular differentiation, are capable of massively parallel operation and (almost) transparent fault tolerance.

References

[1] M. Abramovici, M. A. Breuer, and A. D. Friedman, *Digital Systems Testing and Testable Design*, Computer Science Press, New York, 1990.

[2] M. Abramovici and C. Stroud, "No-overhead BIST for FPGAs", In *Proc. 1st IEEE International On-Line Testing Workshop*, pp. 90-92, 1995.

[3] S. B. Akers, "Binary decision diagrams", *IEEE Transactions on Computers*, c-27(6), June 1978, pp. 509-516.

[4] Stephen D. Brown, Robert J. Francis, Jonathan Rose, Zvonko G. Vranesic, *Field-programmable gate arrays*, Kluwer Academic Publishers, Boston, 1992.

[5] A. Burks, ed., *Essays on Cellular Automata*, University of Illinois Press, Urbana, IL, 1970.

[6] J. Byl, "Self-Reproduction in Small Cellular Automata", *Physica 34D*, pp.295-299, 1989.

[7] E.F. Codd, *Cellular Automata*, Academic Press, New York, 1968.

[8] D. Floreano, "Reducing human design and increasing adaptivity in evolutionary robotics", in T. Gomi, ed., *Evolutionary Robotics*, AAI Books, Ontario, Canada, 1997, pp. 187-220.

[9] F. Hanchek, S. Dutt, "Methodologies for Tolerating Cell and Interconnect Faults in FPGAs", *IEEE Transactions on Computers*, v. 47, n. 1, January 1998.

[10] M. H. Hassoun, *Fundamentals of Artificial Neural Networks*, The MIT Press, Cambridge, MA, 1995.

[11] T. Higuchi, M. Iwata, I. Kajitani, H. Iba, Y. Hirao, T. Furuya, B. Manderick. "Evolvable Hardware and its Application to Pattern Recognition and Fault-Tolerant Systems". In E. Sanchez, M. Tomassini, eds., *Towards Evolvable Hardware*, Lecture Notes in Computer Science, Springer, Berlin, 1996, pp. 118-135.

[12] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory Languages and Computation*, Addison-Wesley, Redwood City, CA, 1979.

[13] W.K. Huang, F. Lombardi, "An Approach for Testing Programmable/Configurable Field Programmable Gate Arrays", *IEEE VLSI Test Symposium*, 1996.

[14] J. R. Koza, F. H. Bennett III, D. Andre, and M. A. Keane, "Automated {WYWIWYG} Design of Both the Topology and Component Values of Electrical Circuits Using Genetic Programming", in *Genetic Programming 1996: Proceedings of the First Annual Conference*, The MIT Press, Cambridge, MA, 1996, pp.123-131.

[15] J. Lach, W.H. Mangione-Smith, M. Potkonjak, "Efficiently Supporting Fault-Tolerance in FPGAs", *Proc. FPGA 98*, Monterey, CA, February 1998, pp. 105-115.

[16] C. G. Langton, "Self-Reproduction in Cellular Automata", *Physica 10D*, pp.135-144, 1984.

[17] D. Mange, D. Madon, A. Stauffer, G. Tempesti, "Von Neumann Revisited: A Turing Machine with Self-Repair and Self-Reproduction Properties", *Robotics and Autonomous Systems*, Vol. 22, No. 1, 1997, pp. 35-58.

[18] D. Mange, M. Goeke, D. Madon, A. Stauffer, G. Tempesti, S. Durand. "Embryonics: A New Family of Coarse-Grained Field-Programmable Gate Array with Self-Repair and Self-Reproducing Properties". In E. Sanchez, M. Tomassini, eds., *Towards Evolvable Hardware*, Lecture Notes in Computer Science, Springer, Berlin, 1996, pp. 197-220.

[19] D. Mange, M. Tomassini, Eds., *Bio-inspired Computing Machines: Towards Novel Computational Architectures*, Presses Polytechniques et Universitaires Romandes, Lausanne, Switzerland, 1998.

[20] P. Marchal, P. Nussbaum, C. Piguet, S. Durand, D. Mange, E. Sanchez, A. Stauffer, G. Tempesti. "Embryonics: The Birth of Synthetic Life". In E. Sanchez, M. Tomassini,

eds., *Towards Evolvable Hardware*, Lecture Notes in Computer Science, Springer, Berlin, 1996, pp. 166-197.

[21] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs*, Springer-Verlag, Berlin, 3rd ed., 1996.

[22] R. Negrini, M. G. Sami, and R. Stefanelli, *Fault Tolerance Through Reconguration in VLSI and WSI Arrays*, The MIT Press, Cambridge, MA, 1989.

[23] A. Perez-Uribe and E. Sanchez, "FPGA Implementation of an Adaptable-Size Neural Network", in *Proc. International Conference on Artificial Neural Networks ICANN96*, Bochum, Germany, July 1996

[24] J.-Y. Perrier, M. Sipper, and J. Zahnd, "Toward a Viable, Self-Reproducing Universal Computer", *Physica 97D*, pp.335-352, 1996.

[25] U. Pesavento. "An Implementation of von Neumann's Self-Reproducing Machine". *Artificial Life*, 2(4), 1995, pp. 337-354.

[26] J.A. Reggia, S.A. Armentrout, H.-H. Chou, Y. Peng, "Simple Systems That Exhibit Self-Directed Replication", *Science*, Vol.259, pp.1282-1287, 26 February 1993.

[27] E. Sanchez, D. Mange, M. Sipper, M. Tomassini, A. Perez-Uribe, A. Stauffer. "Phylogeny, Ontogeny, and Epigenesis: Three Sources of Biological Inspiration for Softening Hardware". In T. Higuchi, M. Iwata, W. Liu, eds., *Proc. 1st Int. Conference on Evolvable Systems: From Biology to Hardware (ICES96)*, Lecture Notes in Computer Science, vol. 1259, Springer-Verlag, Berlin, 1997, pp. 35-54.

[28] M. Sipper, *Evolution of Parallel Cellular Machines: The Cellular Programming Approach*, Springer-Verlag, Berlin, 1997.

[29] C. Stroud, S. Konala, and M. Abramovici, "Using ILA testing for BIST in FPGAs", *Proc. 2nd IEEE International On-Line Testing Workshop*, Biarritz, July 1996.

[30] A. Stauffer, "Membrane building and binary decision machine implementation", Technical Report 247, Computer Science Department, EPFL, Lausanne, 1997.

[31] G. Tempesti, "A New Self-Reproducing Cellular Automaton Capable of Construction and Computation", *Proc. 3rd European Conference on Artificial Life*, Lecture Notes in Artificial Intelligence, 929, Springer Verlag, Berlin, 1995, pp. 555-563.

[32] G. Tempesti, D. Mange, A. Stauffer, "A Robust Multiplexer-Based FPGA Inspired by Biological Systems", *Journal of Systems Architecture: Special Issue on Dependable Parallel Computer Systems*, EUROMICRO, 43(10), 1997.

[33] G. Tempesti. *A Self-Repairing Multiplexer-Based FPGA Inspired by Biological Processes*.Ph.D. Thesis, Swiss Federal Institute of Technology, Lausanne, 1998.

[34] A. Thompson, "Silicon Evolution", in *Genetic Programming 1996: Proceedings of the First Annual Conference*, The MIT Press, Cambridge, MA, 1996, pp. 444-452.

[35] S. Trimberger, ed., *Field-Programmable Gate Array Technology*, Kluwer Academic Publishers, Boston, 1994.

[36] J. von Neumann, *The Theory of Self-Reproducing Automata*, A. W. Burks, ed. University of Illinois Press, Urbana, IL, 1966.

[37] S. Wolfram, *Cellular Automata and Complexity*, Addison-Wesley, Reading, MA, 1994.