Efficient self-replication of digital circuits in programmable logic devices

Joël Rossier, André Stauffer, *Member, IEEE,* and Gianluca Tempesti, *Member, IEEE,*

***Abstract*—This article presents the implementation of a self-replication algorithm in a Field Programmable Gate Array (FPGA). Whereas previous research on self-replication has been mostly limited to theoretical examples and small problem sizes, this work shows how the Tom Thumb self-replication algorithm can be extended to take into account realistic FPGA architectures and representative processor-scale digital logic circuits.**

To achieve this objective, the algorithm was implemented within the POEtic tissue, a programmable logic device for bio-inspired systems, and a dedicated processor, based on the MOVE paradigm, was developed. Starting from a single processor, the self-replication process can be used to generate an arbitrarily large array of identical processors that then differentiate to realize a given application. In this article, this process is demonstrated through a four-processor system that implements a simple counter. The ultimate goal of this research is to demonstrate how a bio-inspired approach can exploit self-replication to tackle the complexity and high fault rates of next-generation electronic devices.

***Index Terms*—Self-replication, FPGAs, bio-inspired hardware.**

I. Introduction

THE self-replication of computing systems is an idea that dates back to the very origins of electronics: in the 1950s, John von Neumann was among the first to investigate the design of processor-scale computing devices capable of self-replication [1] [2] with the goal of obtaining reliability through the redundant operation of many copies of the original device.

Since von Neumann's ground-breaking work, research on self-replicating computing machines has gone through several phases, but, in general, interest in applying self-replication directly to electronic hardware waned because of technological hurdles. In recent years, the introduction of programmable logic devices such as FPGAs has revitalized the field of biologically-inspired hardware by allowing (at least in theory) the run-time modification of hardware. The physical processes that underlie the operation of organisms in nature remain unattainable in electronic devices, but they can be approximated by altering the configuration of a programmable device.

However, practical applications of the self-replication process to electronics remain almost non-existent, probably due to the considerable amount of hardware overhead that is inevitably associated with their implementation.

J. Rossier and A. Stauffer are with the School of Computer and Communication Sciences, École Polytechnique Fédérale de Lausanne (EPFL), CH-1015 Lausanne, Switzerland; e-mail: j.rossier@epfl.ch, andre.stauffer@epfl.ch.

G. Tempesti is with the Department of Electronics, University of York, Heslington, York YO10 5DD, UK; e-mail: gt512@york.ac.uk.

Manuscript received XXXXXX XX, 200X; revised XXXXXX XX, 200X.

On the other hand, some of the motivations that led von Neumann to study self-replication are beginning to re-surface among researchers faced with design and robustness issues in next-generation electronic devices. The vast amount of on-chip resources that will be available in the next few decades, either by further shrinking silicon fabrication processes or by the introduction of molecular scale devices, together with the predicted features of such devices (e.g., high fault sensitivity), will introduce layout and fault-tolerance issues that cannot be solved using current design methodologies [3] [4] [5].

In this context, the usefulness of a self-replication process that allows a complex circuit to automatically replicate within a programmable substrate is fairly obvious:

- as biological organisms *grow* from an initial cell to a complete adult, so large arrays of cellular computing elements could exploit self-replication to grow in the programmable substrate, rather than being completely specified at design time;
- faced with faults in the substrate, a growth process could be able to avoid faulty areas, while the redundancy that is an automatic result of self-replication can potentially allow the circuit to self-repair in the case of online faults.

To demonstrate the feasibility of self-replication in the context of complex electronic circuits, however, it is necessary to take into account several practical issues, such as the hardware overhead and the efficiency of the mechanisms involved, and advance beyond the "toy" examples that have been traditionally used to illustrate this process. This article describes the process whereby a recently-developed algorithm was adapted to implement self-replication within a real-world programmable device and applied to a system consisting of four dedicated processors. While still simple, this system is much more complex than any circuit to which self-replication has been applied to date and exploits mechanisms and architectures that can be easily scaled to larger systems.

After a background on some of the historical approaches to self-replication in section II, section III presents the basic operation of the Tom Thumb Algorithm on a minimal example. The FPGA substrate that has been used to implement the self-replication process is illustrated in section IV and in section V the architecture of the self-replicating processors is defined. Section VI describes the modifications made to the basic Tom Thumb algorithm and to the FPGA to efficiently implement self-replication. Finally, section VII deals with the implementation of the self-replication process. A closing section (VIII) will introduce future directions and a discussion on the possibilities of the proposed approach to self-replication.

II. A short background on self-replication

The self-replication of computing machines has a long history, punctuated by relatively few milestones. The following is a brief outline of the main approaches used to study this process, ranging from von Neumann's original ideas to some of the latest results in the area.

Of course, it should be mentioned that the concept of self-replication has been applied to artificial systems in contexts other than computing. A classic example is the 1980 NASA study by Robert Freitas Jr. and Ralph Merkle [6] (recently expanded in a remarkable book [7]), where self-replication is used as a paradigm for efficiently exploring other planets. However, the self-replication of physical machines rather then computing systems is beyond the scope of this article and this short background will not extend to cover this kind of approaches.

A. Von Neumann's Universal Constructor

Multicellular organisms are among the most reliable complex systems known to man, and their reliability is a consequence not of any particular robustness of the individual cells, but rather of their extreme redundancy. One of the basic mechanisms which provides such reliability is cellular division, i.e., self-replication at the cellular level. Von Neumann, confronted with the lack of reliability of the computing systems he was designing, turned to this mechanism to find inspiration in the design of fault-tolerant computing machines.

In particular, Von Neumann [2] investigated self-replication as a way to design and implement digital logic devices and attempted to develop an approach to the realization of self-replicating computing machines (which he called *artificial automata*, as opposed to natural automata, that is, biological organisms).

Using cellular automata (CA) as a framework, von Neumann realized the first self-replicating system. Based on a 29-state CA, his approach centered on a *Universal Constructor* (a machine capable of building any other machine, given its description) composed of two parts: a *tape* containing the description of the cellular machine to be built and the constructor itself, a complex structure capable of reading the tape and building the corresponding machine. Given a description of itself, the machine was then able to create copies of itself, first *interpreting* the contents of the tape and then *copying* the tape to the new machine. Coupled with a (possibly universal) Turing machine, the approach conceptually allowed the self-replication of computing systems of arbitrary size and complexity (Figure 1).

Never meant to be implemented in actual hardware, von Neumann's Universal Constructor is an extremely complex machine. A recent estimate by W.R. Buckley [8] places the size of the machine (without the Turing machine) at approximately 800K cells, with a 12M cells tape. Obviously, in spite of the considerable theoretical power of this approach, its complexity prevents its use from a practical standpoint.

B. Langton's Loop

A second stage in research on self-replication was opened by C. Langton in 1983 [9]. In order to reduce the complexity

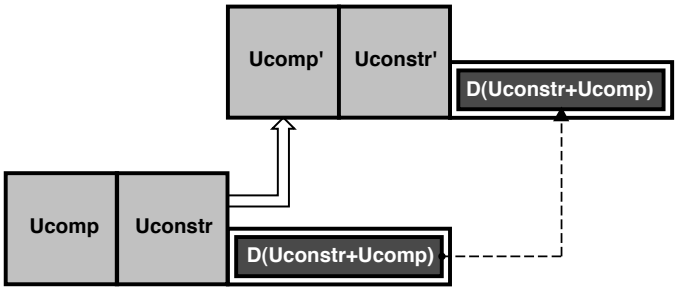


Fig. 1: Von Neumann's Universal Constructor (`Uconst`) can replicate itself and an arbitrary (potentially universal) computing machine (`Ucomp`) given the description `D` of the two machines.

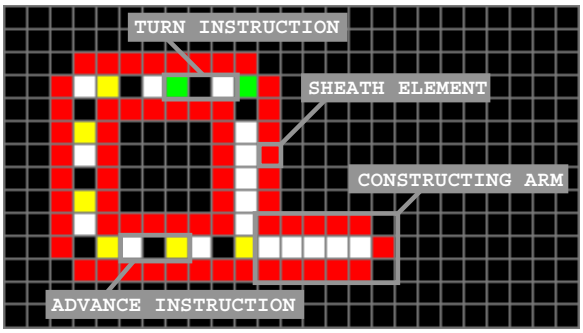


Fig. 2: The initial configuration of Langton's Loop.

of the process, he dropped the universal construction and universal computation ability of the von Neumann system and proposed a simple self-replicating machine in the form of a loop (Figure 2), also implemented as a cellular automaton, based on a constructing arm and on a looping replication program.

Unlike von Neumann, who was interested in self-replication from the standpoint of circuit design, Langton's research was aimed at studying the application of life-like properties to computational structures and his goal in developing his approach was to determine the smallest automaton capable of self-replication. Further improvements to the machine led to smaller versions of the original loop [10] [11] resulting in smaller self-replicating structures, but all these systems, because of the context in which they were studied, lack any computational capability.

More recently, some attempts have been made to redesign Langton's loop in order to embed calculation possibilities. Tempesti's loop [12] is thus a self-replicating automaton, with an attached executable program that is duplicated and executed in each of the copies. Perrier and al. [13] proposed a self-replicating loop showing universal computational capabilities. This system consists of three parts, loop, program, and data, all of which are replicated, followed by the program's execution on the given data. However, the complexity of these approaches, while considerably smaller than von Neumann's Universal Constructor, remains too great to be considered useful in the context of electronic hardware.

C. Other CA-based approaches

All of the approaches mentioned above share the common process of self-replicating through the interpretation of a sequence of building instructions. Some examples of self-replicating CA, however, exploit a different mechanism, that of *self-inspection*: instead of reading and interpreting a description, the self-replicating automaton inspects itself and produces a copy of what it finds. While less general than the universal constructor (obviously, the machine can only build an exact copy of itself), this approach is more versatile than Langton's loop, as structures of (almost) any size and shape can replicate. In practice, however, the best-known example of self-inspection is that of a self-replicating loop [14].

Also, while traditionally there has been a very loose connection between the kind of cellular automata used to study self-replication and actual circuit design, some researchers have been trying to close this gap by studying automata that more closely approach some particular features of digital circuits. An example is Morita and Imai's study of self-replication in the context of reversible cellular automata [15] (in a reversible CA, every configuration has at most one predecessor), inspired by reversible logic in digital circuits.

Similarly, Peper et al. [16] [17] have developed self-replicating structures in Self-Timed Cellular Automata (STCA). This kind of automata do not rely on a global synchronization mechanism to update the states of the cells, but rather the state transitions only occur when triggered by transitions in neighboring cells. The basic assumption in this work is that STCA is a model that might more closely resemble molecular-scale nanoelectronic devices.

D. Self-replication in electronic devices

As seen above, throughout its long history, cellular automata have remained the environment of choice to study how self-replication can be applied to computing systems. However, in general, researchers in the domain (including von Neumann) have never regarded CA as the environment in which self-replication would be ultimately applied. Rather, CA have traditionally provided a useful platform to test the complexity of self-replication at an early stage, in view of eventually applying this process to real-world systems, either to electronics or, more generally, to computing systems.

Approximating a self-replication process in an electronic device, however, required the introduction of programmable circuits, where the physical construction that occurs in nature can be replaced by an information-based process. In practice, self-replication in hardware has been implemented, without exception to our knowledge, as the copy of a partial configuration of a programmable device.

One of the simplest approaches that exploits this kind of setup is *configuration cloning* [18], based on a simple replication of the configuration of part of an FPGA in order to create multiple copies of the same subsystem. In this case, of course, no *self*-replication occurs, since the configuration process is controlled by an external entity. As the external controller still needs to sequentially program the entire circuit, most of the advantages of self-replication are lost.

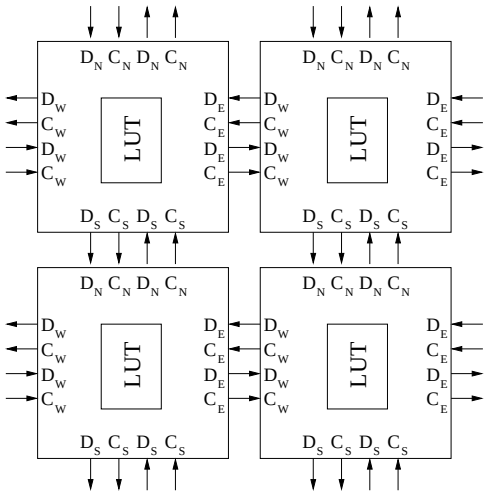


Fig. 3: 2x2 Cell Matrix Grid. Each cell, which has two inputs and two outputs per edge, is connected with its four direct neighbors.

A variation on the same approach, closer to self-replication, was developed by our group in the late 1990s [19] [20], with an emphasis on self-repair. In this system, applied to a very fine-grained custom FPGA, a tiny cellular automaton is used in a first step of the replication process to subdivide the programmable circuit into blocks of arbitrary size (corresponding to the circuit to be replicated). In a second step, an external entity injects a single copy of the configuration of the block, which is automatically replicated so as to completely fill the device. While more versatile than configuration cloning (the automaton does not require external control and the configuration of the device occurs in parallel rather than in series), the approach still requires a relatively complex grid of global connections and an external synchronization, which again limit the advantages of self-replication.

The system that, to date, probably best exploits self-replication in the framework of electronic devices is the Cell Matrix system [21] [22]. Cell Matrix is a fine-grained reconfigurable device composed of a collection of identical elements (referred to as *cells*) placed on the edges of a two-dimensional regular grid. Each cell in the grid is interconnected with its four cardinal neighbors and contains a lookup table, which is used as a truth table to implement logic functions. Each cell can exchange information with its four neighbors, and it uses (as a cellular automaton) its four inputs and its lookup table to define the value of its four outputs.

Unlike the systems presented above that have to be controlled by an external computer in order to achieve self-replication, cells of Cell Matrix circuit can self-replicate autonomously. Each cell is at the same time configurable and can configure other cells without any external input command. This feature is called *self-duality*. In order to implement self-duality, a cell has to operate in two independent modes: *D-mode* and *C-mode*. In D-mode the cell's lookup table processes the four input signals in order to generate output signals, whereas in C-mode the input data is used to fill up (configure) or re-write (re-configure) the lookup table of the cell.

Using self-duality, arbitrarily complex structures can self-replicate within the circuit. Cell Matrix is an accomplished system that has been shown to work well, but is hindered both by the non-conventional structure of the cells (the approach cannot easily be generalized to arbitrary programmable logic architectures) and by the very large overhead required by self-replication, since as in von Neumann's approach the construction process relies on a description of the machine to be built that is in fact much larger than the machine itself.

More recently, another algorithm was proposed to achieve self-replication of arbitrary structures in programmable logic. The Tom Thumb algorithm [23] [24] borrows from Langton and his successors the concept of loop, but was designed to be implemented in silicon. Potentially, the algorithm could be used to replicate any structure within a programmable device, following a simple systematic methodology, but so far its operation has been described only for trivial, illustrative examples. This article shows how the algorithm was extended and applied to a real-world programmable logic device, demonstrating how it can be used for the self-replication of complex circuits.

III. The Tom Thumb algorithm

This section introduces the basic behavior of the Tom Thumb algorithm, designed to enable self-replication in programmable logic. Developed in the context of a more general bio-inspired approach, the *Embryonics* project [20], which ranges from logic gates to massively parallel arrays of processors, the algorithm requires a shift in terminology compared to the traditional CA-based approaches.

In particular, examining in some detail the approaches described in the previous section from the standpoint of biological inspiration, it can be claimed that both the tape of von Neumann's Universal Constructor and the data circulating in the self-replicating loops bear some resemblance to the genome present in every cell in a biological organism. A single instance of the constructor (or loop) can then be seen as a cell, which conflicts with the terminology used in cellular automata, where a cell is the basic element of the array.

Therefore, in the rest of this article, the terms "cell" and "molecule" will be used in a way that corresponds more closely to their biological definitions. The cell will be defined as the smallest part of a living being which carries the complete blueprint of the organism, i.e. the genome, and will represent the unit within the system that can replicate itself. In this case, it will represent a small, dedicated processor. Each cell will then be implemented by an array of "molecules", which in this case represent the basic programmable elements of an FPGA. As will be shown, the Tom Thumb algorithm bears a strong resemblance to a cellular automaton (in fact, the algorithm could be implemented using a conventional CA, but to avoid confusion this terminology will not be used).

To complete the terminology used within the Embryonics approach, the term "organism" indicates a complete computing system composed of several cells working together (i.e., an application-specific array of processors), while the term "population" (not used in this article) refers to a set of several organisms. The complete 4-level hierarchy of complexity is shown in figure 4.

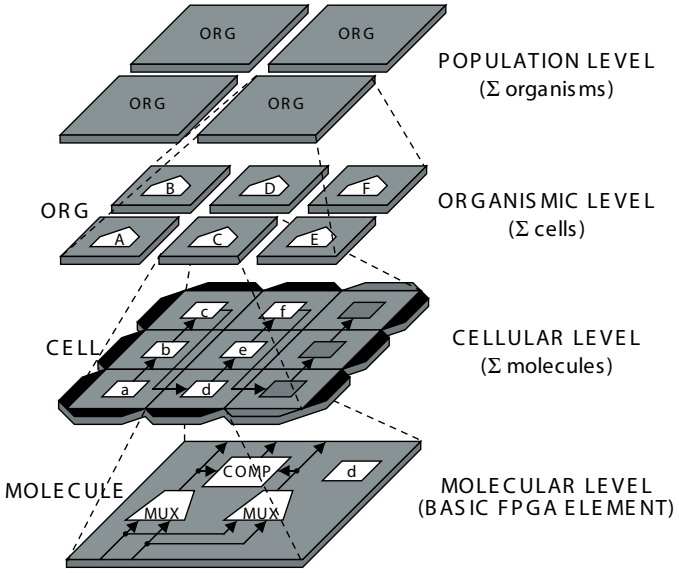


Fig. 4: The four hierarchical levels of the Embryonics approach.

A. Basic structure

The basic operation of the Tom Thumb algorithm can be illustrated by means of a minimal cell composed of four molecules, which grows and then divides to spawn two daughter cells. This simple example is sufficient to define the mechanisms that the basic molecule has to implement to enable the self-replication of the cells.

For this example, a molecule is defined as a basic programmable logic element with no functionality. That is, the element consists simply of a memory to store a minimal configuration, which is not used for any purpose except to store a unique number. Because of the operation of the algorithm, the configuration of the molecules must be stored in a set of shift registers: while obviously more expensive in terms of surface compared to other solutions, this kind of memory storage has the fundamental advantage of allowing data to move easily within the programmable substrate, a necessary condition for a self-replicating process to take place.

The purpose of the self-replication of a mother cell in the context of a programmable logic substrate is to replicate the configuration data of its molecules at a different location within the substrate, creating identical copies of the mother cell. The Tom Thumb algorithm enables such a behavior if the cell and the molecules have the following characteristics: first of all, a *configuration path* has to be defined inside the cell in such a way that the configuration registers of the molecules are connected in a loop that goes through each of the molecules of the mother cell. A path of this type that is valid for a cell composed of four molecules is shown in the figure 5(a). This requirement is also the reason why the minimal cell that can replicate using the algorithm has to be composed of four molecules, organized as a square of two rows by two columns.

The second requirement of the algorithm is that each molecule, in addition to its configuration, must contain a *flag* indicating the direction of the configuration path (the

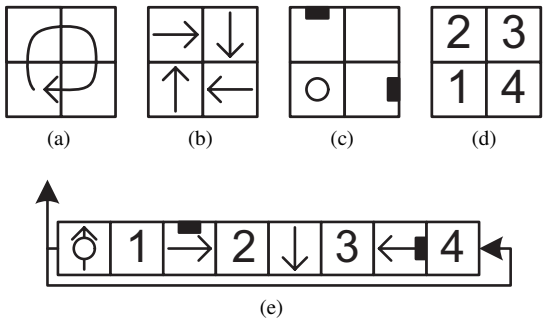


Fig. 5: Basic Tom Thumb information and genome

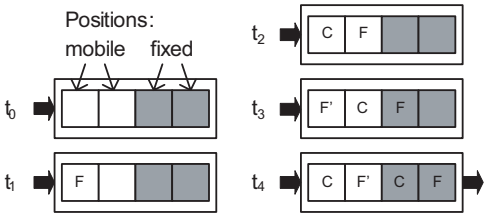


Fig. 6: Configuration of a single molecule and definition of the direction of the next molecule in the configuration path.

arrows in figure 5(b)). Additionally, the flag information must indicate which molecule is placed at the beginning of the loop (arbitrarily defined as the lower left corner), shown with a circle in figure 5(c), and which molecules will be used to send out the data required for the creation of the daughter cells, shown with black rectangles in the same figure.

Taking into account that the configuration of the molecules is defined by the numbers 1 to 4 in figure 5(d), it results that the minimal cell implementing the algorithm is entirely defined by the configuration bitstream shown in figure 5(e). This bitstream, which represents the genome of the cells, is composed of eight *packets*, four of them containing the flags and the other four used to define the configuration data.

Finally, the molecules must be able to store *two* copies of their configuration data and flag. The first copy will be used as the actual configuration, while the second will circulate within the loop and be used to create the daughter cells.

Roughly equivalent to the double-helix configuration of DNA in biological cells, this requirement introduces a considerable amount of overhead, but greatly simplifies the self-replication process (incidentally, a variation of the algorithm, mentioned in the conclusion of the article, does away with this duplication, but introduces other complications).

In the minimal example, since each molecule is defined by two packets of information (the configuration data and the flag), its configuration memory will have to store four packets and, in order to fully configure a cell, the bitstream shown in figure 5(e) will have to be injected twice.

The construction of the first cell, which occurs when the genome is injected into the substrate from an external loader will now be illustrated. Then, an explanation of how this first mother cell duplicates to create copies of itself, i.e. reproduces itself by configuring daughter cells, will be given.

B. Constructing the mother cell

Note that in the genome represented in figure 5(e), the first, third, etc. packets always contain flag information (F in figure 6), while the the second, fourth, etc. packets contain the configuration data of the molecules (C in figure 6). As shown in the same figure, each molecule contains four memory positions able to store the packets. The two positions on the right of the molecule will be used to store the fixed information, i.e. the flag defining the the role of the molecule in the Tom Thumb algorithm and the actual configuration of the molecule, while the two left positions will be used to transmit the packets to the next molecule on the construction path and to contain the second copy of the genome that will be used for the replication process.

At each time step t_x, a packet of the original genome is shifted and injected in the programmable substrate. For practical reasons, the bottom-left molecule in the array is normally considered as the first to be accessed, but of course (the path being a loop) any molecule can be used to start the configuration. As the Tom Thumb algorithm relies entirely on local connections between neighbours, this initial molecule represents the only *injection point* where an external connection is necessary for the configuration of the entire substrate.

When the first, empty molecule receives the first packets, it shifts them until its two fixed positions (on the right in figure 6) are filled. During this process, at time t_3 the molecule becomes aware of which flag (F) will be stored in its fixed position, at which point it can establish a new connection to forward the following packets of the configuration in the direction indicated by the flag in order to configure the next molecule on the path. This connection becomes valid one clock cycle later, i.e. at time t_4. At this moment, the molecule has received its configuration and the flag defining in which direction the construction will proceed and has created the appropriate connection path. All further configuration packets are shifted through the two left memory positions and then out of the molecule in the direction indicated by its fixed flag.

The molecule that receives the configuration packets behaves in the same way and this process repeats itself until each molecule of the cell has been configured. The entire process, for the genome of figure 5(e), is shown in figure 7. At times t_4, t_8, t_{12} and t_{16}, new connections are established between molecules. At time t_{16}, the the configuration loop has closed and the four molecules of the cell are configured.

Note that, during the construction process, the genome has been inserted twice. The first copy is memorized in the two right memory positions of each molecule (*fixed memory*) and configures the molecules. In parallel, the second copy shifts indefinitely through the two left memory positions (*mobile memory*) of the molecules following the configuration path. This second copy of the genome will be used to instantiate the replication of the cell, as described in the next subsection.

Note also that the flag trapped in the fixed memory positions of each molecule recalls the pebbles left by Tom Thumb in the well-known fable to memorize his way, an analogy that gives the algorithm its name.

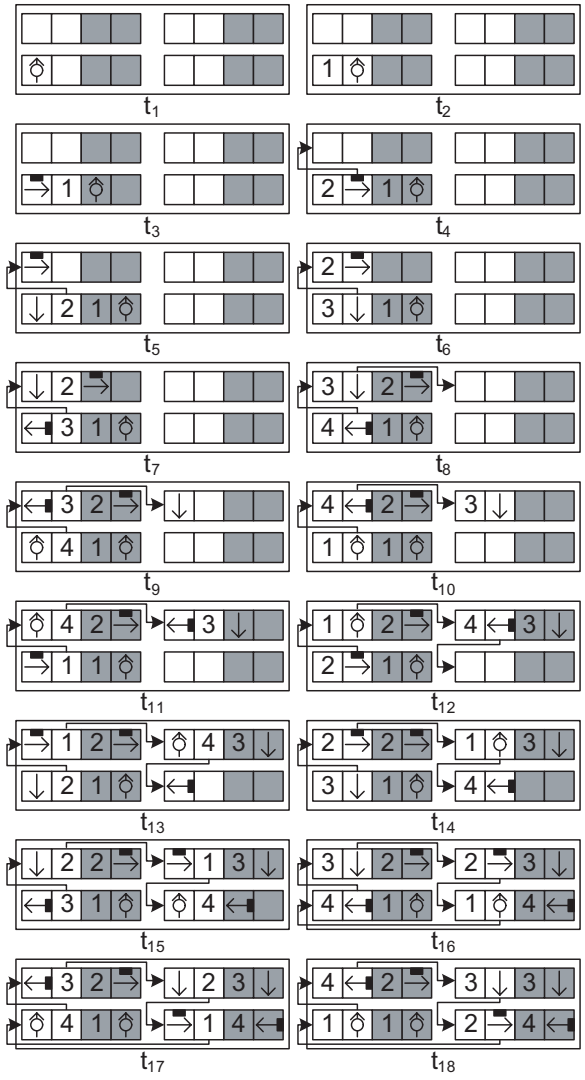


Fig. 7: Construction of the first cell. Connection setup: t_4 north, t_8 east, t_{12} south, t_{16} west

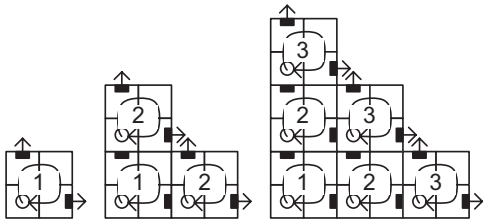


Fig. 8: Pattern of cell replications

C. Self-replication of the mother cell

In order to grow an artificial organism composed of multiple cells, the Tom Thumb algorithm allows a cell to replicate in both horizontal and vertical directions, as shown in figure 8, where cell 1 replicates to construct the cells labeled 2, which in turn replicate to construct the cells labeled 3. As a result, a cell is able to trigger the construction of two daughter cells to its north and to its east.

The first steps of the replication process towards the north in the minimal example are detailed in figure 9. At time t_{11},

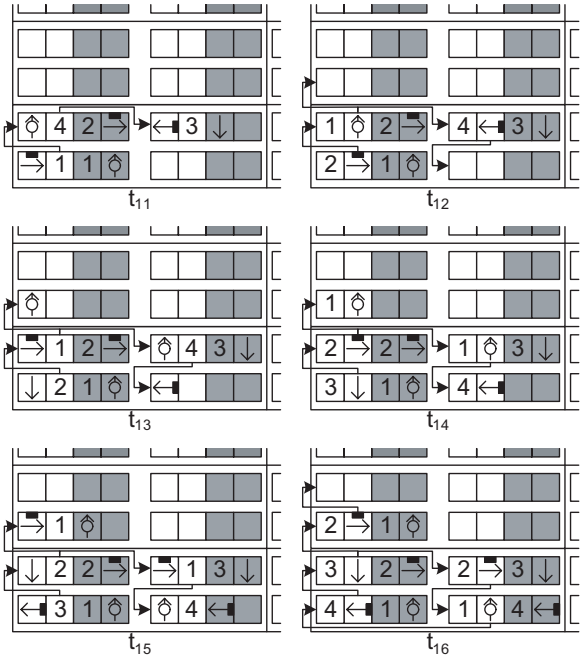


Fig. 9: First steps of the cell replication to the north

the upper left molecule of the mother cell receives the first packet of the configuration. Because it is configured with a flag indicating that it has to launch the replication to the north (the black rectangle in the figure), in addition to the direction of the construction path it establishes a new connection that will be the start of the replication path to the north.

This new connection becomes active at the next clock cycle, i.e. at time t_{12}, and the packets of the configuration, in addition to being shifted to the east following the configuration loop, are also duplicated and shifted to the north. The molecule that receives them will behave exactly as the initial molecule for the construction of the mother cell and begin the construction of the first daughter cell.

Figure 9 shows the configuration of this first molecule and the establishment of the first construction path to the north that will be used to configure the second molecule of the daughter cell. The replication of the mother cell to the east (not shown) follows exactly the same process, but starts at time t_{24}.

Note that after the mother cell has emitted the cell configuration to the north twice, i.e. when it has sent two times the eight packets of the cell genome at time t_{28}, the daughter cell is fully configured and has closed the loop of its construction path. As a result, the replication path to the north can be suppressed and the two cells will become separate (if identical) entities ready to operate independently of each other.

D. Hardware implementation

Based on purely local interactions between neighbours, the Tom Thumb algorithm could be implemented by a canonical CA. However, while the conventional criteria used in the design of CA render them notoriously inefficient to implement in hardware, the algorithm was conceived using a design methodology that privileges ease of implementation. The DSCA (for

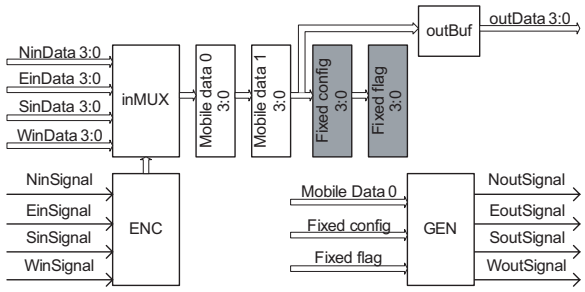


Fig. 10: DSCA implementation of the Tom Thumb algorithm

Data and Signals Cellular Automata) paradigm [25] defines the transitions of an element from one state to the next not by accessing a truth table, but rather as a consequence of a set of data and signals received from the element's neighbours. In particular, at each clock cycle information is sent from each element to its cardinal neighbors, together with a signal that indicates if the information has to be processed.

In the Tom Thumb algorithm, the data that needs to be sent corresponds to the packets, consisting of either a flag or the configuration data for the programmable elements. In practice, the size of the packets is determined by the number of required flags, since the flag information needs to be transmitted within a single packet. For its basic version (the one that is described in this section), the flags that are needed to implement the algorithm are the following: one *empty flag* that corresponds to a non-configured register, four *directional flags* corresponding to the four arrows that are used to create the construction path, one *start flag* indicating the first molecule of the path, and finally two *branching flags* identifying the molecules that handle the replication process. These eight different flags can be coded with three bits. One additional bit is necessary to determine the type of packet that a molecule is receiving, in order to discriminate between flag and configuration packets. This implies that the smallest size for a packet of the basic Tom Thumb algorithm is four bits.

Using the DSCA approach, the hardware design of the basic version of the algorithm is more or less straightforward and is shown in figure 10 (a more detailed description of the implementation can be found in [23]). The size of the busses linking two adjacent molecules has to be at least equal to five (one bit for the DSCA signal and four bits for the packets).

With a packet size of four bits, the maximum number of different configurations of the molecules is limited to eight. Obviously, this number is not sufficient to represent the possible configuration of a real-world programmable logic device: the configuration memory of programmable elements typically ranges from several dozens to hundreds of bits. However, the example used in this section represents the *minimal* loop capable of implementing the Tom Thumb algorithm. In practical applications to a programmable device, the algorithm can be extended both to larger cells (i.e., cells that are composed of an arbitrarily large number of molecules) and, with some minor modifications, to much more complex molecules (i.e., molecules whose configuration consists of an arbitrarily large number of memory positions).

In the next section, the programmable logic device that was used in order to demonstrate the features of the Tom Thumb algorithm in a realistic setting will be described. This custom device, developed for the "Reconfigurable POEtic Tissue" project, funded by the Future and Emerging Technologies programme (IST-FET) for the European Community, was selected both because of the presence of some features that are interesting in the context of bio-inspired systems and, perhaps more importantly, because the hardware architecture of its elements could be freely accessed and modified. It is worth noting, however, that the same algorithm can be similarly adapted to any programmable device architecture, as long as its configuration memory can be structured in the form of a shift register.

IV. The POEtic tissue

The *POEtic tissue* [26] [27] is a reconfigurable circuit that draws inspiration from the multi-cellular structure of complex biological organisms to implement the three main models commonly used in bio-inspired systems [28] [29]: Phylogenesis (P), the history of the evolution of the species through time; Ontogenesis (O), the development of an individual as directed by his genetic code, from its first cell to the full organism; Epigenesis (E), the development of an individual through learning processes. All of these models, to a greater or lesser extent, have been used as a source of inspiration for the development of computing machines (for example, ontogenesis in the Embryonics project or epigenesis in artificial neural networks) but the POEtic tissue is the first hardware substrate dedicated to the implementation of systems that could potentially combine the three axes of bio-inspiration into one single circuit.

Physically, the tissue is composed of two layers (Figure 11): a regular, two-dimensional array of programmable logic elements (molecules) and a *cellular routing layer*. This second layer is also a regular, two-dimensional array and consists of special routing units that are responsible for the (long-distance) communication between the cells (once again, cells are defined as processor-scale circuits implemented in the programmable substrate, according to the hierarchy of Figure 4). The routing layer implements a distributed routing algorithm based on identifiers that allows the creation of data paths between cells at runtime.

Each molecule, as well as each routing unit, is connected to its four cardinal neighbors in a regular structure, also shown in figure 11. Moreover, each molecule can access a routing unit to set up a long-distance connection. In the canonical implementation of the POEtic tissue, each routing unit handles connections from four molecules (i.e., there are one fourth as many routing units as there are molecules in the substrate), but the ratio can be varied depending on the predicted connection density.

As shown in figure 12, a molecule mainly contains a 16-bit look-up table (LUT) and a D flip-flop (DFF); its inputs are selected by a set of multiplexers and its outputs can be routed to any cardinal direction through a switchbox. A molecule possesses 76 bits of configuration that define the content of

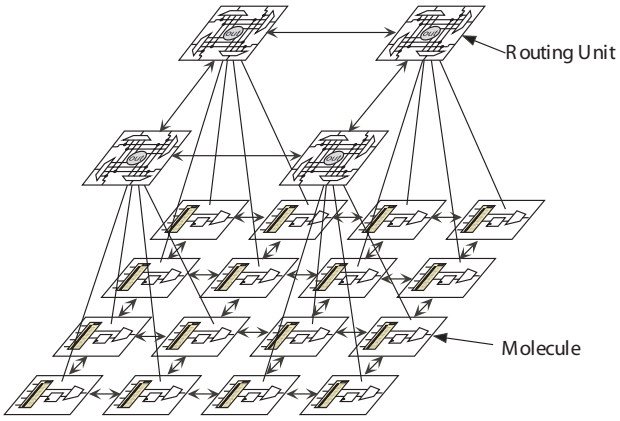


Fig. 11: POEtic two-layer physical structure with the molecules and their routing units.

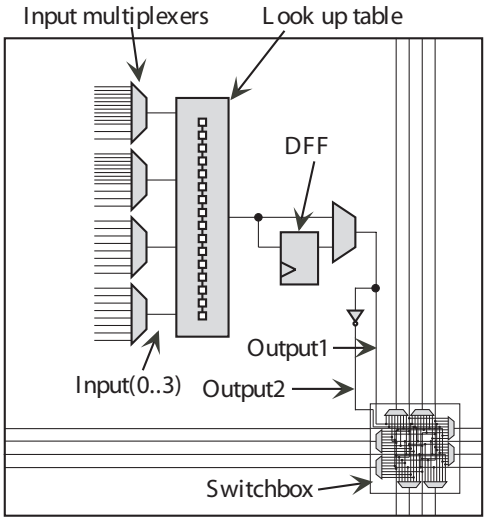


Fig. 12: Basic structure of a POEtic molecule

the LUT and of the DFF, as well as the selection of the multiplexers for the inputs and the outputs of a molecule. Moreover, these configuration bits also select one of the different possible operational modes of a molecule.

The operational modes of a POEtic molecule are a reflection of the diverse requirements of computing and bio-inspired systems. To implement conventional logic designs, it can be configured as a simple 16-bit LUT, as two 8-bit LUT, as a 8-bit LUT plus a 8-bit shift register, or as a 16-bit shift-register. Then there are four additional operational modes that are specific to the POEtic tissue: the first two are the Output and Input modes in which the molecule is connected to its routing unit and contains the 16-bit long routing identifier of the molecule itself, respectively of the molecule from where the information has to arrive (more on this subject below). The third special mode is the Trigger mode, in which the task of the molecule is to supply a trigger signal needed by the routing algorithm for synchronization purposes. The last mode is the Configure mode, in which a molecule has the capability of partially reconfiguring its neighbors, i.e. the molecule can modify a fixed subset of the configuration bits of its neighbors (68 bits out of 76).

Inter-molecular communication, i.e. short-range communication between the programmable logic elements in the POEtic circuit, is implemented by a switch box composed of multiplexers and two directional lines to and from each cardinal direction. This kind of communication is used to implement the gate-to-gate connections required to implement circuits within the programmable substrate.

Inter-cellular routing, i.e. long-range communication between the processors implemented using the programmable logic, is implemented using a distributed routing algorithm, inspired by Moreno [30], that dynamically connects the inputs and outputs of the cells. The connection paths are set up using a parallel implementation of the breadth-first search algorithm, similar to Lee's algorithm, that configures the multiplexers contained within the routing units.

The dynamic routing approach used in POEtic has many advantages compared to a static routing process. First of all, it requires a small number of clock cycles to finalize a path. Secondly, when a new cell is created it can start a routing process without the need of recalculating all the paths already created. Thirdly, a cell has the possibility of restarting the routing process of the entire organism if needed. Finally, this approach is totally distributed, without any global control over the routing process, a clear advantage where scalability is concerned.

The operation and the non-standard features of the POEtic tissue are described in some detail elsewhere [26] [27]. In fact, many of these details are not relevant to this article, since the POEtic tissue was selected as a test platform essentially because of the possibility to modify its hardware structure and not for its detailed architecture. This capability is of course crucial to allow the circuit to be modified to implement the Tom Thumb algorithm. In section VI, the modifications, both to the algorithm and to the POEtic tissue, required to achieve the self-replication of complex circuits, will be discussed.

V. Cellular processors

A simple system based on dedicated processors (representing the cells in the algorithm and in the hierarchy of figure 4) was chosen to verify the functionality and efficiency of the Tom Thumb algorithm applied to complex circuits implemented with the POEtic tissue.

In particular, this section will present a four-processor system whose purpose is quite simple: to count minutes and seconds. As will be shown, the system uses self-replication to generate four identical copies of a processor, which then link together and differentiate in order to execute a specific part of the code, i.e. each processor will be responsible for one of the four digits of the counter.

The next subsection will present the MOVE paradigm, that is, the architecture that was selected to implement the processors. The advantage of this architecture is that it allows to easily design the processors without requiring high-level synthesis tools. Then, in the second subsection, the detailed architecture of the processor, describing its functional units and the way they are linked, will be outlined.

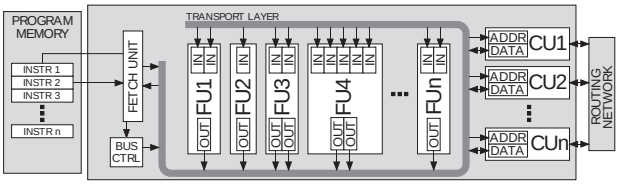


Fig. 13: Basic architecture of a MOVE processor

A. The MOVE paradigm

The MOVE paradigm, also known as the Transport-Triggered Architecture [31] [32] [33], was originally developed for the design of application-specific dataflow processors (processors where the instructions define the flow of data, rather than the operations to be executed).

In many respects, the overall structure of a MOVE system is fairly conventional and the basic differences lay in the architecture of the processor itself, and hence in the instruction set.

Rather than being structured, as is usual, around a more or less serial pipeline, a MOVE processor (Figure 13) relies on a set of Functional Units (FUs) connected together by one or more transport busses. All the computation is carried out by the functional units (examples of such units can be adders, multipliers, register files, etc.) and the role of the instructions is simply to move data from and to the FUs in the order required to implement the desired operations. Since all the functional units are uniformly accessed through input and output registers, instruction decoding is reduced to its simplest expression, as only one instruction is needed: `move`.

The `move` instructions trigger operations which, in the simplest case, correspond to normal RISC instructions. For example, in order to add two numbers a RISC `add` instruction has to specify two operands and, most of the time, a destination register to store the result. The MOVE paradigm requires a slightly different approach to obtain the same result: instead of using a specific `add` instruction, the program moves the two operands to the input registers of a functional unit that implements the add operation. The result can then be retrieved from the output register of the functional unit and moved wherever it is needed.

The reasons for choosing the MOVE paradigm for the processors are two-fold: on one hand, its compactness and versatility makes it ideally suited to the bio-inspired systems studied, while on the other hand its simplicity allows to define the layout of the systems on the POEtic tissue, for which no automated synthesis tools exist.

As the contents of the LUTs, the connectivity between the molecules, the structure of the functional units and that of the communication units that implement the connection network of the cellular array have to be defined by hand, there was a necessary limitation to a very simple system. However, the array remains sufficiently complex to be a good illustration of the scalability of the self-replication approach.

B. Architecture of the system

As mentioned above, the final system (the organism) will be composed of four MOVE processors (the cells) that will form a 4-digit counter that will display seconds and minutes (in practice, two connected modulus-60 counters). Each of the processors will handle one digit. Thus, two will count from 0 to 9 while the two others will count from 0 to 5. In their final configuration, they are logically organized to form a chain that is represented in the organismic level of figure 14.

Shown in the same figure, the Seed Unit is used to start the counter as soon as the replication of the processors is finished and to provide the first (rightmost) processor of the final chain the information it needs to launch the *differentiation process* outlined in section VII. This process is an important part of the setup of an organism within the bio-inspired approach: self-replication generates multiple copies of an identical processor, while differentiation determines the precise role of each of the copies within the organism. Several kinds of approaches can of course be used to implement this process (see, for example, [34] for a partial survey), but for the purposes of this article it can be seen as a local mechanism whereby each cell determines its own position within the organism and, as a function of this information, determines which instructions to execute (in this precise case, whether to count to 5 or to 9).

The operation of the final system is rather obvious: the processor that handles the rightmost digit, i.e. the units of seconds, permanently counts from 0 to 9. When this processor arrives at 9, it generates a signal (`EnableCount`) telling the next processor, which handles the tens of seconds, to increment its own digit. When the tens of seconds processor arrives at 5, it generates in turn a signal enabling the next processor on the chain, i.e. the units of minutes, to count. Again, once this processor reaches 9, it signals the next processor to count the tens of minutes.

As exposed in the precedent section, the processors were realized using the MOVE paradigm. Clearly, the system is rather trivial, but the objective of the exercise was to test the implementation of self-replication in a real implementation and not to design a multi-processor system for performance. The advantage of using MOVE processors in this context is that the behavior of the system could easily be extended to more complex applications by redesigning the functional and communication units without in any way altering the self-replication algorithm.

C. Implementation in the POEtic tissue

The actual implementation of a processor in the POEtic tissue is shown in the cellular level of figure 14, while its architecture is detailed in figure 15. The processor contains the following Functional Units:

- `CMP` and `INC`, used to compare or increment a set of internal values (addresses, conditions, etc.)
- `EN`, which receives the `EnableCount` signal from the preceding processor in the chain and sends the same signal to the next processor at the appropriate time.
- `IOprec` and `IOnext`, responsible for the dynamic setup of the connections between the processors in order to propagate the position (and hence the function) of each processor within the system, the `EnableCount` signals and the other control signals required for the operation

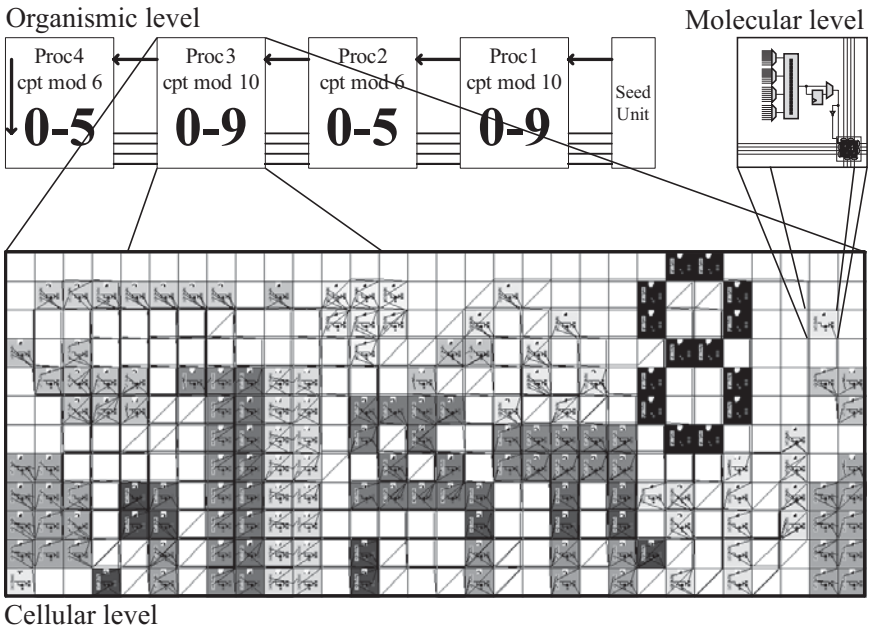


Fig. 14: The three hierarchical levels of the system (cf. Figure 4): the organism implementing a counter, the cell mapped on the POEtic tissue, and the molecule.

of the counter. Through these FUs, the processor accesses and controls the behavior of the Input and Output molecules, i.e. it can force a molecule to establish a connection or allow a molecule to accept the connections through the dynamic connection network of the POEtic tissue.

- `POS`, used to compute the position of the processor inside the system (and hence its function within the organism), as a function of the data received from the `IOprec` FU.

Additionally, each processor, as is usual in the MOVE paradigm, contains a data bus, spanning all the FUs, and two memory busses: one for the source addresses and the other for the destination addresses of each `move` instruction.

The processors rely on two separate internal memories: the first (`DCMem`) contains the code for the differentiation and connection mechanisms, while the second (`MEM`) stores the instructions for the normal operation of the processor (time counting and signal generation).

As the processors were realized on the POEtic substrate, which provides a specific molecular mode to implement shift registers, it was decided that, instead of an addressable memory that could support jumps in the code, *cyclic memories* [35] would be used, where each instruction is read successively, and executed or not, depending on the special unit called Execution Stack (`EXok` in figure 15).

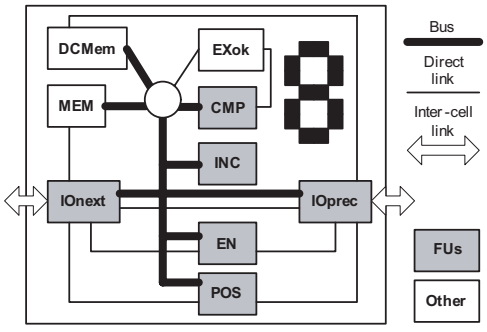


Fig. 15: Detailed architecture of the processor.

To summarize the behavior of the Execution Stack, it can be said that, when facing an *"if condition then (x1; x2; ...) else (y1; y2; ...) end"* instruction, if the condition is valid, the stack will permit the execution of the X instructions and then block the Y instructions. Otherwise, it will block the X execution and permit the Y instructions (a more detailed explanation of this unit can be found in [36] [37]). The stack generates a signal that drives the `MemToBus` part of the circuit (the white circle in figure 15) which can enable or not the instructions and data presented by the two memories to be transmitted on the processor busses.

Finally, for demonstration purposes, a special unit (represented by the `8` in figure 15) was added, used to display the digit stored in each processor. This unit, which would not be useful in an actual electronic implementation within an integrated circuit, was introduced in conjunction with a dedicated simulation setup to visualize the internal operation of the circuit through a custom GUI. The results of this visualization will be used in section VII to illustrate the self-replication of the processors.

While trivial in many respects, the system described above represents probably the most complex circuit to which self-replication has ever been applied. Orders of magnitude larger than the minimal example provided in section III to illustrate the behavior of the Tom Thumb algorithm, its implementation required several minor modifications both to the basic algorithm and (to a much lesser extent) to the basic structure of the POEtic elements. These modifications will be described in some detail in the next section.

VI. Implementation issues

Section III presented the basic operation of the Tom Thumb algorithm. It is clear that the algorithm, in its minimal form, cannot be applied to a real design. Several modifications to the basic algorithm are necessary to allow it to implement self-replication in a real-world programmable logic device and to increase its efficiency from the standpoint of hardware resources.

In particular, the following aspects of the basic algorithm have been addressed:

- Only three bits of useful information are available for the configuration of a molecule. It is obvious that a real system must handle a greater number of configuration bits for its basic component, i.e. the molecules.
- A cell can only replicate in two directions, i.e. to the north and the east. While in some cases this might be sufficient, in a real system it would be an advantage to replicate in the four cardinal directions.
- Five bits are transmitted from a molecule to its neighbors in each clock cycle. This number could be changed, which would have an impact on the size of the busses that link two adjacent molecules and change the number of clock cycles needed for the replication.
- The cell has no control on the replication process, which is launched at startup and continues until the substrate is entirely configured. The algorithm could be modified to enable the cells to choose when and where to replicate.
- There is a large overhead in terms of registers needed for the replication, as every bit has to be duplicated in the static genome and in the running configuration. While much of this redundancy is unavoidable, some optimizations are possible to reduce the number of duplicated bits.
- Each molecule in the replicated cell is activated as soon as its static configuration has been set. In a real circuit, this can lead to strange behaviors as parts of the circuit become active before the rest is configured.

Strictly speaking, only the first of these issues is crucial to achieve self-replication in an FPGA. The rest are, essentially, performance issues that increase the efficiency or the versatility of the self-replication approach. In this section, the modifications made to the basic algorithm to address these issues will be detailed.

In addition to the modifications to the algorithm required for an implementation in the POEtic tissue, the latter also required some minor alterations to efficiently implement the self-replication of the organism described in section V. These alterations will be summarized in the last part of this section.

A. Size of the configuration data

In the basic Tom Thumb algorithm exposed in the section III, the configuration data was coded in only one group of three bits, implying that the maximum amount of information usable to configure a molecule is limited to three bits. It is clear that in a real application, these three bits would not be sufficient to define the configuration of a molecule. For example, applying the algorithm to the POEtic tissue, a molecule would be the equivalent of an FPGA element and would require 76 bits of configuration. Generally speaking, the bits required to configure a LUT-based FPGA element (for example, at least 16 for a 4-input LUT and several more to define the behaviour of the interconnection network and of the register) are many more than the three at disposal with the basic algorithm.

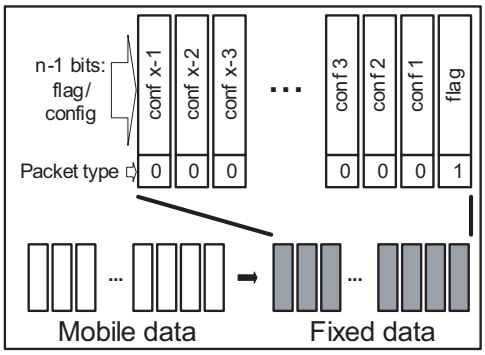


Fig. 16: Configuration memory for a single molecule.

Luckily, the Tom Thumb algorithm was designed specifically with this option in mind and it is relatively straightforward to increase the size of the configuration memory within the algorithm.

In the Tom Thumb algorithm, in each clock cycle a molecule receives a packet of n bits ($n = 4$ in the basic algorithm). Among these bits, one, the *packet type* bit, is used to determine if the other $n-1$ bits are flag or configuration information. In the basic version, the entire information needed by a molecule is received in two n-bit packets, one for the flag and the other for the configuration data.

In order to cope with a greater number of configuration bits, the algorithm was modified by multiplying the number of packets used for the configuration data (Figure 16). In this new design, the data needed for one molecule that uses c configuration bits and a flag coded with f bits require $x = \lceil (c+f)/(n-1) \rceil$ different n-bit packets consisting of one bit indicating the packet type and $n-1$ bits for the information (flag or configuration). Assuming the minimal packet size, i.e. $n = f+1$ bits as shown in the figure, the total number packets becomes $x = \lceil c/(n-1) \rceil + 1$.

This simple modification allows the algorithm to replicate a cell composed of molecules requiring any number of configuration bits. Incidentally, this same modification reduces somewhat the resource overhead of the algorithm, since only one flag packet is necessary for each molecule, whatever the number of configuration data packets.

B. Direction of self-replication

The basic Tom Thumb algorithm is designed to enable the cell replication in only two directions, i.e. to the north and to the east of the mother cell. Assuming that the injection point for the first configuration is placed in the south-west corner of the array, this restriction does not affect the basic operation of the system.

However, taking into account the possibility of alternate (or indeed multiple) injection points, or the application of the self-

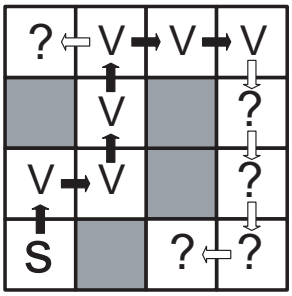


Fig. 17: Replication with defaults

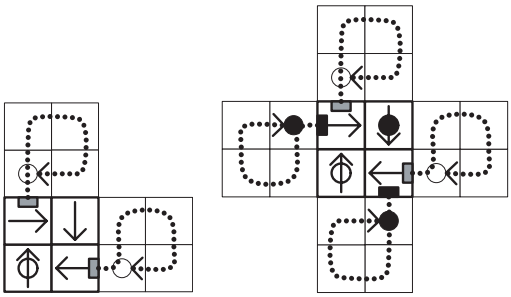


Fig. 18: Basic and modified TTA directions of replication

replication process to a faulty substrate, then the two directions are not sufficient to guarantee an optimal replication pattern within the programmable circuit.

Fault tolerance, in particular, is one of the main motivations to justify the need for a self-replication process: assuming that the programmable substrate in which self-replication occurs can contain faults (a more than reasonable assumption in the kind of electronic or nanoelectronic devices that are the main targets of this approach), then it is necessary for the algorithm to be able to avoid faulty areas of the circuit and replicate only in the fault-free areas.

In figure 17, each square represents the area occupied by a full cell (composed of many basic molecules). The bottom left square is the initial cell that starts the replication process, i.e. the mother cell, and the gray squares are areas that contain faulty molecules and should be avoided when replicating.

With the basic two-directional algorithm, the replication process will only be able to make copies of the mother cell in the squares marked with a 'V', following the path shown with the black arrows. The squares labeled with question marks will not be configured, and the area will be wasted.

To avoid this potential loss of resources, the Tom Thumb algorithm has been extended to enable replication in the four cardinal directions. This extension will allow the entire fault-free surface of the circuit to be exploited, whatever the location of the faulty areas (for the example of figure 17, replication will follow the configuration path indicated by the white arrows).

As detailed in section III, the Tom Thumb algorithm works using a path spanning all the molecules of the cell to be replicated, a flag marking the first molecule on this path and two more flags indicating the molecules from which the replication has to occur and its direction. These are shown in figure 18(a): the direction of the path in each molecule is represented by the straight arrows, the start flag is the empty circle and the flags for the directions of the replication are the small gray rectangles. The dotted arrows show the new paths constructed during the replication process.

In order to make a cell replicate in four directions, the basic algorithm (Figure 18(b)) was modified by adding a new flag and modifying two existing ones. The basic concept of the path spanning all the molecules of the cell, defined by the four basic directional flags, remains identical. The two branching flags were modified to indicate the two additional directions for the replication (these are shown with the small black rectangles in the figure). Finally, while keeping the original start flag to indicate the first molecule in the path, a second start flag (the black circle in the figure) had to be introduced to indicate the first molecule in the path for the new directions of replication: for the replication to the north and to the east directions, the first molecule to be configured is the one located at the bottom left of the cell, while for the west and the south directions the replication starts with the molecule located at the upper right corner of the cell.

Adding the empty flag, the number of flags used for the replication in the four directions is now equal to nine. This implies that the minimum packet size for this modified version of the algorithm becomes five bits (four bits for the flag plus one bit for the packet type).

It should be noted that, to allow the Tom Thumb algorithm to implement self-replication in a faulty substrate, the ability to replicate in the four directions is a necessary, but not sufficient condition. Obviously, this capability needs to be coupled with a mechanism that allows the algorithm to recognize that an area within the circuit is faulty, so as to avoid it when replicating. This requirement has led to a re-design of the Tom Thumb algorithm to implement this testing function [38]. This novel version (not used for the purposes of this article), while more complex than the "standard" version, remains quite similar and the same procedure that was followed to adapt the standard algorithm to a real-world FPGA can be followed for the new version.

C. Size of buses

Another modification that can be made to the basic self-replication algorithm, one that does not affect the algorithm itself but rather its implementation, is to vary the size of the busses used to link the molecules in the configuration path.

In the basic algorithm, five bits must be transmitted from one molecule to the next: one bit is used for the signal transmission and four bits for the data packet (one bit for the packet type and three bits for the actual configuration information). Obviously, while the signal and packet type bits cannot be altered, the width of the configuration data that is transmitted at every clock cycle can be parameterized, keeping in mind that the flag information has to be transmitted in a single clock cycle and that, as a result, this width needs to be at least three, respectively four bits, for a replication in two, respectively four (see subsection VI-B), directions.

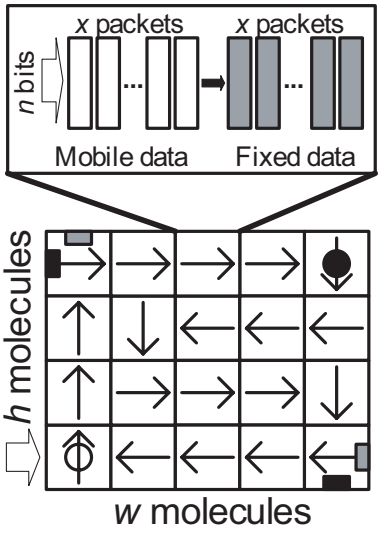


Fig. 19: Typical path spanning a $5 * 4$ cell.

As is often the case in this kind of scenarios, the selection of the bus size gives rise to a compromise between size and speed. In this particular case, increasing the size of the busses requires, obviously, a larger amount of hardware resources, in the form of physical wires connecting the molecules (wires that, however, are purely local from one molecule to the next, and hence less costly than long-distance connections). On the other hand, it can considerably reduce the number of clock cycles required for the replication of a cell.

Assuming that a cell has a width of w molecules and a height of h molecules, a packet has a size of n bits, each molecule has a configuration of c bits and the flag is coded with f bits, then, as detailed in subsection VI-A, a molecule must be able to store $x = \lceil (c+f)/(n-1) \rceil$ packets in its fixed positions. Obviously, the mobile data requires x additional packets positions.

To illustrate the effects of changing the bus size, some timing calculations will be presented for a configuration path spanning a cell composed of $w*h$ molecules ($5*4$ in figure 19). In this path, the starting molecule is at the bottom left corner of the cell. The path first goes straight to the north, then to the south, going successively in the east and west direction, and finally comes back to the west from the bottom right corner.

For an injection of the configuration in the substrate starting at time 0, at time $t_1 = 2 * x$ the first molecule will have fixed its entire configuration, and the second molecule will have received its own configuration at time $t_2 = 4 * x$. After $t_{cell} = 2 * w * h * x$ clock cycles, the first cell is entirely configured.

The replication to the north starts when the flag of the starting molecule (the white circle in the figure) arrives at the top left molecule. The second copy of the configuration is injected in the circuit at time $t_{secondconf} = w * h * x$ and therefore is forwarded to the north from the top left molecule at time $t_{startnorth} = t_{secondconf} + h * x = (w + 1) * h * x$. It results that the cell to the north will be fully configured at time $t_{cellnorth} = t_{startnorth} + t_{cell} = (3 * w + 1) * h * x$. A similar calculation indicates that the cell to the east is fully configured after $t_{celleast} = (4 * w * h - w + 2) * x$ clock cycles.

As each of these timing depend on the x value that is a function of the packet's width n, by altering the size of the busses, the replication time can be changed. It is minimal when $x = 1$, i.e. when the flag and the whole configuration are stored in one single packet. The ratio $c/(x + f)$ between the number of useful configuration bits c and the bits used expressly for the replication process (x bits to define the packet types and f bits for the flag) could also be modified.

D. Startup vs. runtime replication

In the basic Tom Thumb algorithm, the configuration of the cell is injected inside the circuit at startup: the first cell (bottom left in the examples) is configured and then the replication process is automatic, filling the entire surface available on the substrate. The cells are duplicated again and again as long as there are free, not-yet-configured molecules. In such a set, the cells have no influence on the replication process, i.e. they cannot decide when and where to self-replicate.

While the creation of a full array of identical processors at startup can have several practical advantages in a number of systolic applications, from a more general point of view it would be useful to give the cells the ability to choose when and where to start their replication. This capability would add the versatility, for example, to host more than one type of cell (and hence more than one organism) within the substrate, or allow a cell to create temporary copies of itself when executing a computationally intensive task, and destroy them when the task is over.

To increase the versatility of the Tom Thumb algorithm, then, a possible modification would be to change the way the cells decide to replicate. In the classic algorithm, described in section III, a replication signal is emitted every time the start flag cell arrives in a molecule configured (with a branching flag) to initiate the replication (the grey and black rectangles in figure 19.

A simple alteration to the algorithm would consist of disabling, by default, these replication signals. A cell would then have the ability to enable them, individually to replicate in only one direction or together to replicate in all directions at once, whenever it decides to create a copy of itself.

In such a system, as in the basic algorithm, the running genome would constantly turn within the cell following the configuration path. When the start flag arrives in a molecule that could initiate the replication, if the latter has not been enabled by the cell, nothing happens. On the other hand, when the cell decides to replicate, it enables one of the molecules that can initiate the replication, which, when the start flag arrives, generates a replication signal and duplicates the genome in the chosen direction to create another cell, as in the original algorithm.

With this modification, the cell can now decide *when* to replicate by deciding when to enable the molecules that initiate the replication. Moreover, it can also choose *where* to replicate, by enabling only the desired replication direction in the appropriate molecule.

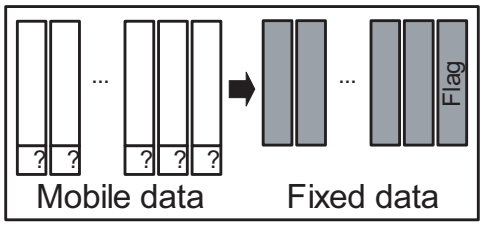


Fig. 20: Suppression of packet type bits in the fixed memory positions.

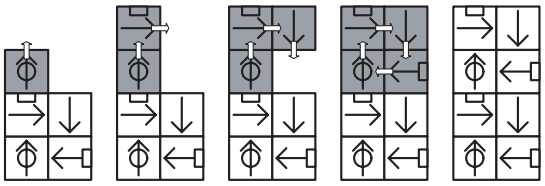


Fig. 21: Disabled molecules during replication process

E. Bit optimization

There is no escaping the fact the Tom Thumb algorithm implies a large area overhead, and as such it is suited for the kind of circuits where the advantages of self replication for layout or fault tolerance are more important than resource optimization. As a consequence, while it is possible to considerably reduce the hardware required for the algorithm through a series of more or less complex alternative implementations (e.g., by using self-inspection instead of keeping a second copy of the configuration data [39]), for the purpose of illustrating the operation of the algorithm in this article this kind of optimizations were limited to a small simplification that does not affect the operation of the algorithm.

Observing the implementation of the Tom Thumb algorithm shows that the packet type bits are only used during the construction process. Their value in the mobile data is used to indicate when to start the duplication of the genome and generate replication signals. Once the data has been memorized in the fixed positions within the molecule, this information becomes useless, and the packet type bits can be suppressed from the fixed data registers without in any way altering the algorithm (Figure 20). This simple modification allows to reduce the size of the configuration memory of each molecule by $x + 1$ bits (where x is the number of configuration data packets).

F. Disabling molecules during cell replication

In the basic Tom Thumb algorithm, during the replication process, as soon as a molecule has been configured, it starts to operate according to the configuration data it has received. While not an issue for the function-less device used to illustrate the operation of the algorithm in section III, such a behavior could potentially be dangerous when the algorithm is applied to a real programmable logic device. In practice, the result of this process would be the step-by-step activation of parts of the processor (the logic gates implemented by each molecule) while the rest is still waiting to be configured. To achieve correct functionality, every molecule of the entire cell should start their normal processing at the same time, once the entire cell has been configured.

To achieve this behavior, an additional 1-bit output in each direction was added to each molecule. Then, as shown in figure 21, when the first molecule of the cell is configured, it generates an active signal through this output in the direction of the replication path (the white arrow in the figure). This signal is forwarded combinatorially by each newly configured molecule and disables the normal functionality of the molecules that receive it. Following the configuration path, the signal is forwarded to every molecule of the replicated cell and disables them as soon as they are configured.

At the end of the replication, when the last configured molecule of the cell closes the path, it signals that the replication process is finished. As a result, the disable signal is deactivated and this information is forwarded through the entire replication path, enabling all the molecules of the cell to start their normal functionality at the same time.

G. Modifications to the POEtic tissue

Because the Tom Thumb self-replication algorithm affects only the configuration of any FPGA it is applied to, the two modifications required to adapt the POEtic tissue to the algorithm are minor and only concern the setup of the configuration memory.

The first modification implied a re-design of the architecture of the configuration memory to fit the structure of the algorithm. As shown in the figure 16, the data injected in a molecule has to be divided into several separate slots that are chained together as a shift register. As a result, the configuration registers of the POEtic molecules had to be modified so that they can be set by shifting data packets of $n-1$ bits. Note that, having suppressed the bits for the packet type in the fixed memory (subsection VI-E), the registers were sized accordingly.

The second modification to the POEtic molecule concerns the ability for the system to decide when and where to replicate (subsection VI-D). Even if the test system (described in section V, as well as in the next section) will not take advantage of runtime replication, this ability was nevertheless added to the substrate. As a result, the molecules had to be altered slightly to allow the functional part to control the configuration memory in order to enable the replication process. This implied the creation of a new operational mode for the POEtic molecule. In this mode, when a molecule contains one of the two replication flags of the Tom Thumb algorithm, the value of the LUT inputs can be used to enable or not the replication in a specific direction.

In addition, independently of the Tom Thumb algorithm, the POEtic tissue had to be modified slightly to allow the implementation of the test system described in section V.

The main modification required for the implementation of the counter was an upgrade of the IO modes of the POEtic molecules: in the basic specification, the molecules set in the Input or Output modes, representing the source or the target of

a dynamic routing path, have a single control signal that forces or not a connection to be established. To efficiently implement the differentiation of the processors after the self-replication phase, this aspect of the POEtic architecture was modified by adding to the IO molecules a second control signal that forces the molecule to accept or not a connection. As a result, the new version of the POEtic IO molecules has two control signals: one used to force a molecule to establish a connection and the other to accept them.

A second, and final, modification was the introduction of an additional input to each molecule, driven directly by the user. This input is, in reality, required neither for the Tom Thumb algorithm nor for the actual counter. Rather, it is related to the user interface of the simulation platform used for the implementation of the system: quite simply, the additional input enables the user to act directly on the state of the flip-flop of each molecule, giving the possibility to interact with the system to activate some of its parts (for example, it is used to launch the Seed Unit described in section V-B). Its purpose is purely to allow the user to control the timing of the simulation.

VII. Self-replication of cellular processors

With the modifications outlined in the previous section, the Tom Thumb algorithm can be applied to the POEtic tissue to instantiate the self-replication of complex circuits.

This section will describe the operation and implementation of the system outlined in section V-B, i.e. how a MOVE processor can replicate itself on a POEtic tissue using the Tom Thumb Algorithm in order to implement a basic multi-processor time counter.

A. Parametrization of the system

The modifications applied to the Tom Thumb algorithm and the POEtic tissue in the preceding section introduced a series of parameters and options that need to be specified for the implementation of the final system.

- The option of disabling the molecules while self-replication occurs (subsection VI-F) was implemented to avoid spurious effects.
- The self-replication in four directions (subsection VI-B) was implemented, even if it is not used in the specific example chosen to illustrate the algorithm, where no fault injection mechanism was introduced.
- The cells can control their replication at runtime (subsection VI-D), but again this feature was not used in the example because it is not required by the application.

Data busses with a width of $n = 5$ bits were used: one bit for the packet type transmission and the other four bits for the configuration data or the flag. This solution represents the minimal width as the flag data must be expressed with at least four bits (9 flags for the replication in four directions, as described in the subsection VI-B).

Since a POEtic molecule is fully defined by 76 configuration bits, $x = 76/(5-1) = 19$ memory slots are needed for the packets of the fixed configuration, plus one additional slot that will be used for the flag data. Another $x+1$ slots are required to store the running data. As a result, each molecule needs $2x + 2 = 40$ clock cycles to be fully configured.

The bits indicating the type of the packets in the static slots were suppressed (subsection VI-E), enabling to decrease by 20 bits the size of the registers in each molecule.

A last parameter that had to be set concerns the density of the routing units in the POEtic tissue (section IV): whereas in the basic POEtic connection setup shown in the figure 11 each routing unit is simultaneously connected to four molecules, the POEtic implementation used for the test system contains one routing unit per molecule. This increased density simplifies the simulation and avoids irrelevant congestion issues, allowing a denser connection pattern.

B. System configuration

At startup, the programmable substrate, i.e. the modified POEtic tissue, contains only a small set of configured molecules implementing the seed unit. This unit is located at the right side of the tissue, as shown in the figure 22(a) (the figure displays the state of the circuit through an interface described in subsection VII-D). The first step of the configuration process is the injection of the configuration of the first processor (the mother cell), in the format required by the Tom Thumb algorithm, inside the circuit. The injection point was arbitrarily set at the bottom left molecule of the tissue.

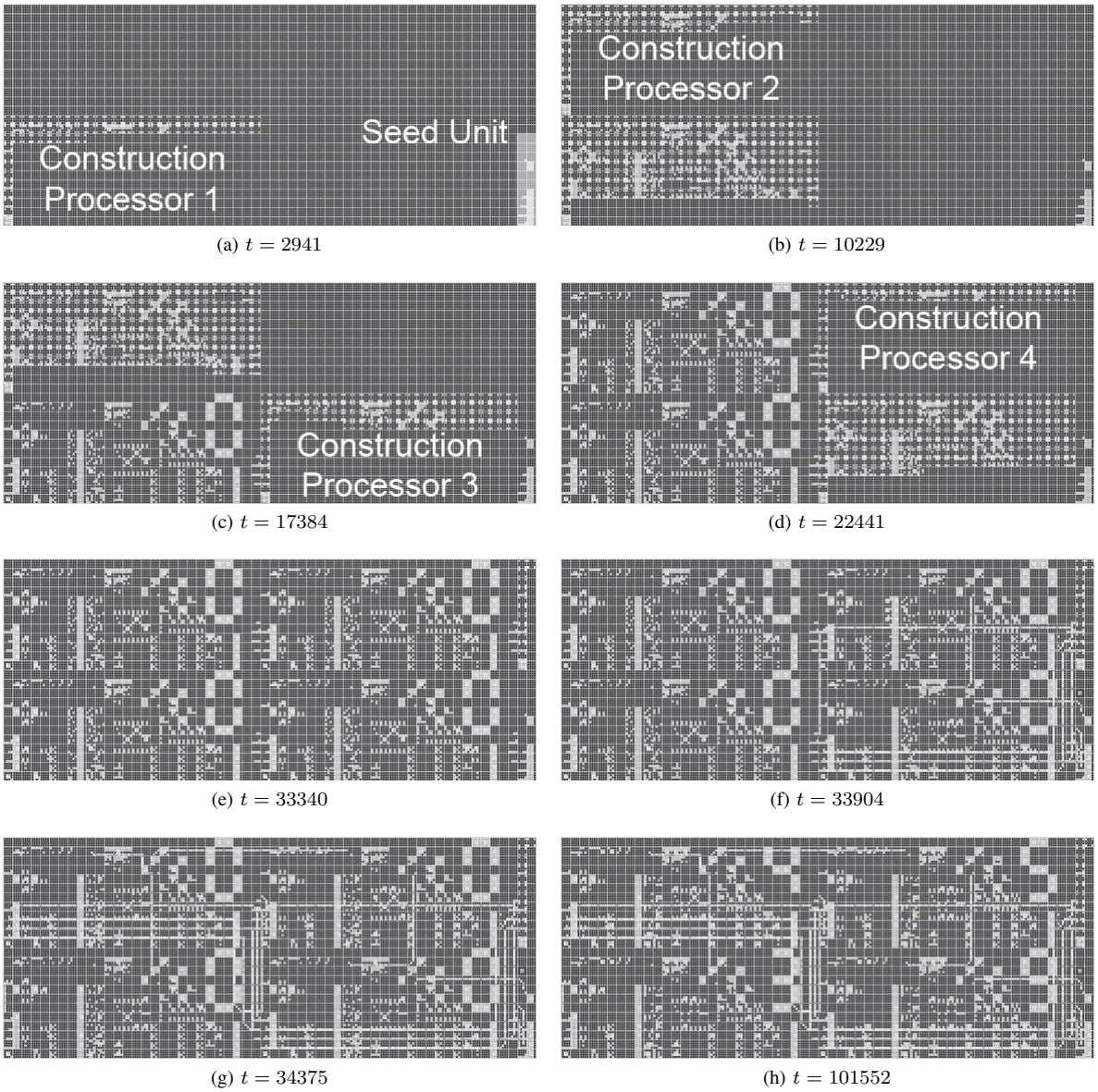
A few steps of the construction and replication processes (which is essentially identical to those of the basic algorithm, illustrated in section III, expanded to take into into account the additional configuration packets) are shown in the figure 22. In the figure, t represents the number of clock cycles from the beginning of the process, i.e. from the injection of the first packet into the circuit.

The injected configuration automatically defines the configuration path and fixes the static configuration of each of the molecules of the first processor to be implemented in the substrate. Then, since the option of using automatic replication at startup (rather than letting the cells decide when and where to replicate) was selected, as explained in subsection VII-A, the replication process is automatically activated in every direction.

As a result, even before this first processor is fully configured, the replication of the genome according to the Tom Thumb algorithm starts to configure a copy of the processor to the north, as shown in figure 22(b). The third processor to be configured is then also a replication of the first one to the east (see figure 22(c)).

Finally, a fourth processor is configured on the substrate as a replication of the second one 22(d). Note that the fourth processor also tries to make a copy of itself in the east direction (figure 22(e)), but this process halts automatically because there are not enough empty molecules to contain a whole copy of the circuit.

When the four processors are configured, the whole system finds itself in an idle state (figure 22(e)): the Tom Thumb algorithm has fully replicated the processors and fixed their configuration.



(a) $t = 2941$ (b) $t = 10229$

(c) $t = 17384$ (d) $t = 22441$

(e) $t = 33340$ (f) $t = 33904$

(g) $t = 34375$ (h) $t = 101552$

Fig. 22: State of the circuit at different time steps for the test system.

C. System operation

At the end of self-replication, the processors wait for an activation signal, telling them to start their differentiation and to connect in the appropriate pattern. As a result, nothing happens until the user launches these processes by activating the seed unit in order to connect to the first processor and transmit the activation signal: in the test system, the seed unit is activated by the user through the additional input to the POEtic molecules described in subsection VI-G. This solution was adopted to improve the demonstration of the system, but obviously several other options are available in a real-world setting (e.g., instructing the seed unit to wait for a pre-determined number of clock cycles before connecting).

When activated, the seed unit connects to the first available processor using the dynamic routing network of the POEtic tissue. In practice, because of the routing algorithm, the seed unit will connect to the closest processor which has not yet been linked, i.e., the processor situated in the bottom right of the circuit in figure 22(e). When connected, the seed unit will send the first activation signal to the processor.

As a result of the activation, the processor will begin to execute the instructions stored in its Differentiation and Connection Memory (`DCMem` in figure 15): it will first setup its `IOprec` FU in order to establish all the connections to the seed unit needed in order to receive the `EnableCount` signal (see subsection V-B), the information that allows it to determine its position inside the chain of processors of the final system (position 0 in this case), and to be able to transmit another signal `sBack` that will be described below. Once again, these connections are established using the dynamic routing network of the POEtic tissue.

Having connected to the seed unit and retrieved the necessary information, the processor knows its position inside the final multi-processor system (position 0) and can continue executing the instructions in the `DCMem` memory.

Since, based on its position, the processor knows that it is not the last one in the system, it sets up its `IOnext` FU in order to connect to another available processor and activate it. When the second processor receives its activation signal, it also starts to execute the instructions in the `DCMem` memory, links to the preceding processor, and determines its position (1) within the final system chain (figure 22(f)).

This process repeats itself until the four processors are activated and linked together as shown in figure 22(g). The fourth processor on the chain, upon seeing that it is the last processor of the organism (position 3), does not try to connect to another processor, but instead activates its `sBack` signal.

This signal propagates back along the chain of processors through the dynamic connections and instructs the processors to execute their `MEM` code, i.e. to begin their normal operation as defined in subsection V-B: depending on the position within the organism, defined through the process described above, the processors *differentiate* and execute different parts of the program stored in the `MEM` memory (processors at positions 0 and 2 count to 9, processors at positions 1 and 3 count to 5). A snapshot of the state of the system after it has counted 34 minutes and 59 seconds is shown in figure 22(h).

D. Implementation

The VHDL specification of the POEtic tissue that had been developed for the EU project was used as a starting point to implement the test system. The modifications detailed in section VI-G were added to the basic architecture of the POEtic molecule. Then, the VHDL code defining the configuration subsystem of the tissue was further modified to implement the Tom Thumb algorithm, as described in section VI. This resulted in the final specification of the molecules used to implement the multi-processor system with self-replication abilities: a matrix of $58*24 = 1392$ such molecules is required to allow the processors (implemented within a rectangular array of $28 * 12 = 336$ molecules) to replicate four times.

Because of the size of the system, a physical implementation in hardware proved impossible. To verify its operation, then, the whole system was simulated using synthesizable VHDL in the ModelSim™environment (several smaller versions of the circuit were physically implemented in an FPGA to verify the transition from simulation to hardware).

In order to observe more clearly the details of the operation of the system during simulation and to be able to control its timing and setup, a custom graphical interface was developed and linked with the ModelSim simulator. This interface is able to display, in the form of an array of colored squares, some of the data from a running simulation. It is also able to start, stop and advance the simulation for a given number of clock cycles. The pictures in figure 22 are screen-shots of the interface.

Finally, the whole system has been synthesized with Leonardo Spectrum using the scl05u library. The results are summarized in the figure 23, showing the amount of hardware resources needed for each part of the system. As each configuration register of the POEtic molecule has to be duplicated in the block implementing the Tom Thumb algorithm, with the addition of the registers storing the flags and other information related to the algorithm process, the overhead in terms of DFFs is obviously high (the DFFs have to be more than doubled). On the other hand, in term of logic gates, the overhead remains quite low, i.e. only 14.8% of the total logic needed for the entire system is used to implement the self-replication process.

	DFFs	%	Gates	%
TT	104	50.5	534	14.8
Routing	26	12.6	796	22
POEmol	76	36.9	2280	63.2

Fig. 23: Hardware requirements of the different parts of the circuit (TT: Tom Thumb algorithm; Routing: dynamic routing layer; POEmol: molecule of the POEtic tissue).

VIII. CONCLUSION

The work presented in this article represents, to the best of our knowledge, the first example of self-replication of a processor-scale digital circuit to be actually implemented in a real-world setting.

This first solution is obviously open to amelioration in several respects and the Tom Thumb algorithm is constantly being improved to address other issues or environments:

- To reduce the overhead inherent in storing a second copy of the configuration in each molecule, a variant of the algorithm has been developed and implemented, which uses a self-inspection approach (subsection II-C) to generate a copy of the configuration whenever needed. Fairly similar to the basic algorithm in its replication dynamics (flags, etc.), this version drastically alters the differentiation and system dynamics.
- Fault tolerance is a crucial application area for self-replication. A self-repairing version of the Tom Thumb algorithm, capable of reconfiguring the programmable array to avoid faults, has been developed [40] and investigations on fault detecting logic are under way. Coupled with the four-direction replication approach described in this article, this version is able to tolerate considerable numbers of faults in the substrate.
- In the context of next-generation electronics, and particularly in that of molecular-scale approaches, the global synchronization required by a cellular automaton (such as Tom Thumb) can be extremely difficult to achieve. To avoid this potential obstacle to the implementation of the algorithm, an asynchronous version has been realized, where the local interactions between the molecules take place without the need for a global clock signal.
- One of the promises of molecular electronics (which could potentially allow a quantum leap in circuit density) is the possibility of being able to design and build circuits that exploit the three dimensions. Because self-replication is an approach designed to simplify layout and design in very complex circuits, a 3D version of the algorithm was designed and implemented [41] [42]. Indeed, because of its structure and its local interactions, the transition of the algorithm from two to three dimensions is quite simple and preserves all its properties.

Several issues connected to the practical implementation of self-replicating processors also need to be addressed: seen as part of a complex bio-inspired design approach, self-replication must be integrated within a complex multi-cellular computing system. This context implies that several "accessory" issues have to be considered:

- The self-replication approach must be extended to real-world applications. While the test system described in this article is sufficient to show that the Tom Thumb algorithm can be applied to structures of arbitrary complexity, the methodology used in the design of the system (or rather, its lack, since the processors had to be designed by hand) does not scale to larger systems. Current work is under way to define a design environment for the synthesis of cellular processors [43].
- The design environment will have to be extended to take into account and exploit the possibilities introduced by self-replication, such as the ability of cells to replicate at will (subsection VI-D), and to integrate fault tolerance and reconfiguration processes in the design.
- The differentiation process of the cellular array will have to be closely integrated in the design as well, to allow newly-replicated cells to seamlessly connect to the rest of the array. The *ad-hoc* approach used in the test system was designed specifically for the given example, whereas a more general solution must be found to implement this process in a wider range of applications.

But even if improvements are of course possible, the self-replication approach described in this article represents a clear step beyond existing solutions, in terms of hardware efficiency and versatility. This latter aspect is particularly important: while the test system used to illustrate the algorithm exploits some of the non-standard features of the POEtic tissue (notably, its dynamic connection network), these features are used exclusively during its operation and are completely disjoint from the self-replication algorithm. In other words, the only requirement of the algorithm is that it must be possible to implement the configuration memory of the programmable device as a shift register. In turn, this implies that the algorithm is relatively technology-independent and can easily be adapted to any programmable structure, as long as the above condition is met, a crucial versatility in view of its implementation in the next generation of electronic devices.

REFERENCES

[1] W. Asprey, *John von Neumann and the Origins of Modern Computing*. Cambridge, MA: The MIT Press, 1992.

[2] J. Von Neumann, *Theory of Self-Reproducing Automata*. Urbana, IL: University of Illinois Press, 1966, edited and completed by A. W. Burks.

[3] K. E. Drexler, *Nanosystems: Molecular Machinery, Manufacturing and Computation*. New York, NY: John Wiley, 1992.

[4] J. Han, J. Gao, Y. Qi, P. Jonker, and J. Fortes, "Toward hardware-redundant, fault-tolerant logic for nanoelectronics," *IEEE Design and Test of Computers*, vol. 22, no. 4, pp. 328–339, 2005.

[5] S. Cotofana, A. Schmid, Y. Leblebici, A. Ionescu, O. Soffke, P. Zipf, M. Glesner, and A. Rubio, "Conan - a design exploration framework for reliable nano-electronics," in *Proc. 2005 IEEE Int. Conf. on Application-Specific Systems, Architectures and Processors (ASAP'05)*. IEEE Computer Society, 2005, pp. 260–267.

[6] R. A. Freitas Jr., T. J. Healy, and J. E. Long, "Advanced automation for space missions," in *Proc. 7th Int. Joint Conf. on Artificial Intelligence (IJCAI81)*. Morgan Kaufmann, 1981, pp. 803–808.

[7] R. A. Freitas Jr. and R. C. Merkle, *Kinematic Self-Replicating Machines*. Georgetown, TX: Landes Bioscience, 2004.

[8] W. R. Buckley and A. Mukherjee, "Constructibility of signal-crossing solutions in von Neumann's 29-state cellular automata," in *Proc. 2005 Int. Conf. on Computational Science (ICCS2005)*, ser. LNCS, vol. 3515. Springer Verlag, 2005, pp. 395–403.

[9] C. G. Langton, "Self-reproduction in cellular automata," *Physica D*, vol. 10, pp. 135–144, 1984.

[10] J. Byl, "Self-reproduction in small cellular automata," *Physica D*, vol. 34, pp. 295–299, 1989.

[11] J. A. Reggia, S. L. Armentrout, H.-H. Chou, and Y. Peng, "Simple systems that exhibit self-directed replication," *Science*, vol. 259, pp. 1282–1287, 1993.

[12] G. Tempesti, "A new self-reproducing cellular automaton capable of construction and computation," in *Advances in Artificial Life: Proc. 3rd European Conf. on Artificial Life (ECAL95)*, ser. LNCS, vol. 929. Srpinger Verlag, 1995, pp. 555–563.

[13] J.-Y. Perrier, M. Sipper, and Z. J., "Toward a viable, self-reproducing universal computer," *Physica 97D*, pp. 335–352, 1996.

[14] J. Ibanez, D. Anabitarte, I. Azpeitia, O. Barrera, A. Barrutieta, H. Blanco, and F. Echarte, "Self-inspection based reproduction in cellular automata," in *Proc. 3rd European Conf. on Artificial Life (ECAL95)*, ser. LNCS, vol. 929. Springer Verlag, 1995, pp. 564–576.

[15] K. Morita and K. Imai, "Self-reproduction in a reversible cellular space," *Theoret. Comput. Sci.*, vol. 168, pp. 337–366, 1996.

[16] F. Peper, T. Isokawa, N. Kouda, and N. Matsui, "Self-timed cellular automata and their computational ability," *Future Generation Computer Systems*, vol. 18, no. 7, pp. 893–904, 2002.

[17] Y. Takada, T. Isokawa, F. Peper, and N. Matsui, "Universal construction and self-reproduction on self-timed cellular automata," *International Journal of Modern Physics C*, vol. 17, no. 7, pp. 985–1007, Feb. 2006.

[18] S. R. Park and W. Burleson, "Configuration cloning: Exploiting regularity in dynamic DSP architectures," in *Proc. ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. ACM Press, 1999, pp. 81–89.

[19] G. Tempesti, "A self-repairing multiplexer-based FPGA inspired by biological processes," Ph.D., Ecole Polytechnique Fédérale de Lausanne (EPFL), 1998.

[20] D. Mange, M. Sipper, A. Stauffer, and G. Tempesti, "Towards robust integrated circuits: The embryonics approach," *Proceedings of the IEEE*, vol. 88, no. 4, pp. 516–541, 2000.

[21] L. Durbeck and N. Macias, "The cell matrix: an architecture for nanocomputing," *Nanotechnology*, no. 12, pp. 217–230, 2001.

[22] N. Macias and P. Athanas, "Application of self-configrability for autonomous, highly-localized self-regulation," in *Proc. 2007 NASA/ESA Conf. on Adaptive Hardware and Systems (AHS2007)*. IEEE Computer Society Press, 2007, pp. 397–404.

[23] D. Mange, A. Stauffer, E. Petraglio, and G. Tempesti, "Self-replicating loop with universal construction," *Physica D*, vol. 191, pp. 178–192, 2004.

[24] D. Mange, A. Stauffer, L. Peparolo, and G. Tempesti, "A macroscopic view of self-replication," *Proceedings of the IEEE*, vol. 12, no. 92, pp. 1929–1945, 1992.

[25] A. Stauffer and M. Sipper, "The data-and-signals cellular automaton and its application to growing structures," *Artificial Life*, vol. 10, no. 4, pp. 463–477, 2004.

[26] A. Tyrrell, E. Sanchez, D. Floreano, G. Tempesti, D. Mange, J.-M. Moreno, J. Rosenberg, and A. Villa, "POEtic tissue: An integrated architecture for bio-inspired hardware," in *Proc. 5th Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES2003)*, ser. LNCS, vol. 2606. Springer Verlag, 2003, pp. 129–140.

[27] Y. Thoma, E. Sanchez, J.-M. Moreno Arostegui, and G. Tempesti, "A dynamic routing algorithm for a bio-inspired reconfigurable circuit," in *Proc. 13th Int. Conf. on Field-Programmable Logic and Applications (FPL03)*, ser. LNCS, vol. 2778. Springer Verlag, 2003, pp. 681–690.

[28] E. Sanchez, D. Mange, M. Sipper, M. Tomassini, A. Pérez-Uribe, and A. Stauffer, "Phylogeny, ontogeny, and epigenesis: Three sources of biological inspiration for softening hardware," in *Proc. 1st Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES96)*, ser. LNCS, vol. 1259. Springer Verlag, 1997, pp. 35–54.

[29] M. Sipper, E. Sanchez, D. Mange, M. Tomassini, and A. Perez-Uribe, "A phylogenetic, ontogenetic, and epigenetic view of bio-inspired hardware systems," *IEEE Trans. on Evolutionary Computation*, vol. 1, no. 1, pp. 83–97, 1997.

[30] J.-M. Moreno, E. Sanchez, and J. Cabestany, "An in-system routing strategy for evolvable hardware programmable platforms," in *Proc. 3rd*

NASA/DoD Workshop on Evolvable Hardware (EH01). IEEE Computer Society, 2001, pp. 157 – 166.

[31] H. Corporaal and H. Mulder, "MOVE: A framework for high-performance processor design," in *Proc.1991 Int. Conf. on Supercomputing*, 1991, pp. 692–701.

[32] H. Corporaal, *Microprocessor Architectures – from VLIW to TTA*. John Wiley & Sons, 1998.

[33] D. Tabak and G. J. Lipovski, "MOVE architecture in digital controllers," *IEEE Transactions on Computers*, vol. C-29, no. 2, pp. 180–190, Feb. 1980.

[34] G. Tempesti, D. Mange, E. Petraglio, A. Stauffer, and Y. Thoma, "Developmental processes in silicon: An engineering perspective," in *Proc. 2003 NASA/DoD Conference on Evolvable Hardware (EH-2003)*. IEEE Computer Society Press, Los Alamitos, CA, 2003, pp. 255–264.

[35] L. Prodan, G. Tempesti, D. Mange, and A. Stauffer, "Biology meets electronics: The path to a bio-inspired FPGA," in *Proc. 3rd Int. Conf. on Evolvable Systems: From Biology to Hardware (ICES2000)*, ser. LNCS, vol. 1801. Springer Verlag, 2000, pp. 187–196.

[36] H. Restrepo, "Implementation of a self-repairing universal Turing machine," Ph.D., Ecole Polytechnique Fédérale de Lausanne (EPFL), 2001.

[37] H. Restrepo, G. Tempesti, and D. Mange, "Implementation of a self-replicating universal Turing machine," in *Alan Turing: Life and Legacy of a Great Thinker*, C. Teuscher, Ed. Berlin, DE: Springer, 2003, pp. 241–269.

[38] A. Stauffer, D. Mange, and G. Tempesti, "Bio-inspired computing machines with self-repair mechanisms," in *Proc. 2nd Int. Workshop on Biologically-Inspired Approaches to Advanced Information Technology (Bio-ADIT06)*, ser. LNCS, vol. 3853. Springer Verlag, 2006, pp. 128–140.

[39] J. Rossier, Y. Thoma, P.-A. Mudry, and G. Tempesti, "Move processors that self-replicate and differentiate," in *Proc. 2nd Int. Workshop on Biologically-Inspired Approaches to Advanced Information Technology (Bio-ADIT06)*, ser. LNCS, vol. 3853. Berlin, DE: Springer Verlag, 2006, pp. 328–343.

[40] A. Stauffer, D. Mange, and J. Rossier, "Design of self-organizing bio-inspired systems," in *Proc. 2007 NASA/ESA Conf. on Adaptive Hardware and Systems (AHS07)*. IEEE Computer Society, 2007, pp. 413–419.

[41] A. Stauffer, D. Mange, E. Petraglio, and F. Vannel, "DSCA implementation of 3D self-replicating structures," in *Proc. 6th Int. Conf. on Cellular Automata for Research and Industry (ACRI04)*, ser. LNCS, vol. 3305. Springer Verlag, 2004, pp. 698–708.

[42] A. Stauffer, D. Mange, E. Petraglio, and G. Tempesti, "Self-replication of 3D universal structures," in *Proc. 6th NASA/DoD Workshop on Evolvable Hardware (EH04)*. IEEE Computer Society, 2004, pp. 283–287.

[43] G. Tempesti, P.-A. Mudry, and G. Zufferey, "Hardware/software co-evolution of genome programs and cellular processors," in *Proc. 1st NASA/ESA Conf. on Adaptive Hardware and Systems (AHS'06)*. IEEE Computer Society, 2006, pp. 129–136.

Joël Rossier has a BSc/MSc in Communication Systems from Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland, obtained in 2004. His master thesis at the Logic Systems Laboratory (LSL) focused on a new type of cellular automaton. He joined the LSL-CARG in May 2004 for a PhD thesis. He is currently working on the hardware application of bio-inspired concepts, particularly on the development of self-replicating processors.

André Stauffer (S'68-M'69) received the M.S. and Ph.D. degrees from the Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland. He spent one year as a Visiting Scientist at the IBM T.J. Watson Research Center, Yorktown Heights, NY, in 1986. He is Senior Lecturer in the School of Computer and Communication Sciences at the Swiss Federal Institute of Technology. He is a Professor at the HES-SO University of Applied Sciences in Yverdon, Switzerland. In addition to digital design, his research interests include cellular automata, circuit reconfiguration, and bio-inspired systems.

Gianluca Tempesti received a B.S.E. in electrical engineering from Princeton University in 1991 and a M.S.E. in computer science and engineering from the University of Michigan at Ann Arbor in 1993. In 1998 he received a Ph.D. from the Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland with a thesis on the design of fault-tolerant bio-inspired FPGAs. In 2003 he was granted a young professorship award from the Swiss National Science Foundation (FNS). He joined the Department of Electronics at the University of York as a Reader in Intelligent Systems in 2006. His research interests include bio-inspired digital hardware, built-in self-test and self-repair, programmable logic, and cellular automata, and he is author of over 75 articles in these areas.