Implementation of a Self-Replicating Universal Turing Machine

Hector Fabio Restrepo[1,2], Daniel Mange[1], and Gianluca Tempesti[1]

[1] Logic Systems Laboratory, Swiss Federal Institute of Technology
CH–1015 Lausanne, Switzerland
E-mail: {name.surname}@epfl.ch
[2] Grupo de Percepción y Sistemas Inteligentes
Escuela de Ing. Eléctrica y Electrónica Universidad del Valle
E-mail: farestre@eiee.univalle.edu.co

Summary. The goal of this contribution is to describe how a universal Turing machine was embedded into a hardware system in order to verify the computational universality of a novel architecture. This implementation was realized with a multi-cellular automaton inspired by the embryonic development of living organisms. In such an architecture, every artificial "cell" contains a complete copy of the description of the machine, a redundancy that allows the introduction of the properties of self-repair and self-replication. These properties were coupled with a modified version of the W-machine to realize a robust, self-replicating universal computer in actual hardware.

1 Introduction

In the 1930's, before the advent of digital computers, several logicians (Kurt Gödel, Alonzo Church, Stephen Kleene, Emil Post, and Alan Mathison Turing) began to think about the theoretical limits of computation. Alonzo Church and Alan Turing independently arrived, through different approaches, at equivalent conclusions. Both solutions described computability, but while Church (1932-34) described it with λ-calculus, Turing's idea (1936) was based on a mathematical model of a machine that could compute any computable function: the *Turing machine* [20,3].

Throughout the history of computer science, the Turing machine has remained a vital benchmark in the validation of novel architectures. Most notably, in the historical work of John von Neumann in the 1950's [22], the computational universality of the Turing machine was coupled with another fundamental property: *constructional universality*, that is, the ability to construct any kind of machine, given its description. The most remarkable product of the coupling of these two properties was the development of the machine known as *Von Neumann's Universal Constructor*, essentially a cellular automaton capable of self-replication and of realizing a universal Turing machine.

Although the complexity of von Neumann's implementation makes it unsuitable for a hardware implementation (indeed, even a software simulation

remains almost beyond the possibilities of modern computer systems), the self-replication of computing machines remains an interesting solution to the problem of realizing "perfect" systems from imperfect components. In fact, the predicted introduction of extremely complex systems realized at the molecular level through, for example, nanotechnology processes is bringing this issue to the leading edge of research.

The contents of this contribution are the result of our research in the domain of computing machines inspired by the properties of biological organisms within a project called *Embryonics* (for "embryonic electronics"). In particular, we shall describe an approach based on the development of complex machines (our artificial organisms) implemented by a *multi-cellular architecture*. In this architecture, organisms are two-dimensional arrays of cells, where each cell is a small processing unit that stores a complete copy of the machine's genome in the form of a microprogram. Each cell then executes only a specific part of the program (the cell's *gene*) depending on its spatial position within the organism (a mechanism analogous to cellular differentiation in living beings), as determined by a set of coordinates.

We have shown in the past that our architectures are capable of implementing the properties of self-replication and of self-repair [6]. To demonstrate the computational universality of our machines we were naturally led, as was von Neumann fifty years ago, to show that our architecture can realize a universal Turing machine [13–15]. Unlike von Neumann, however, we will go beyond pencil-and-paper and implement our self-replicating Turing machine in hardware, demonstrating not only the feasibility but also the efficiency of our approach. The goal of this contribution is the description of our implementation of such a machine.

In Section 2 we present a brief introduction to the concept and structure of Turing machines, from the specialized, application-specific machines to the universal version that is the main topic of this contribution. In Section 3 we introduce the basic features of Embryonics architectures, based on multicellular arrays of cells, along with the architecture of an ideal and of an actual universal Turing machine able to self-replicate exploiting the features of the Embryonics machines. In Section 4 we then describe the PICOPASCAL language and a hardware implementation of a PICOPASCAL interpreter, necessary in order to understand our Embryonics implementation. Section 5 describes the detailed implementation of a self-replicating universal Turing machine. A discussion of our results follows in the final section (Section 6).

2   Turing Machines

2.1   Specialized Turing Machines

In his 1936 paper [21], A. M. Turing defined the class of abstract machines that now bear his name: a *specialized Turing machine* (Figure 1), or simply a *Turing machine*, is a finite-state machine (the *program*, to use Turing's
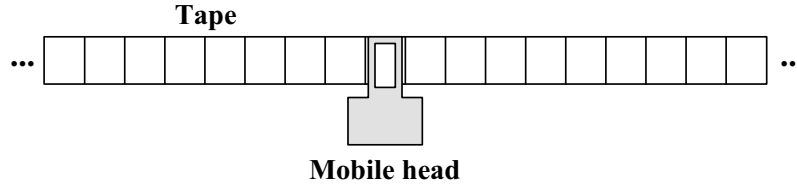
Tape

Mobile head

Fig. 1. A specialized Turing machine

terminology) controlling a mobile head, which operates on a tape. The tape, composed of a sequence of locations (rectangles in the figure), contains a string of symbols (the *data*). The tape should theoretically be considered as infinite in both directions. However, for all practical purposes, we can assume that, when the machine starts operating, the tape will be blank except for some finite number of squares. With this assumption, we can consider the tape as finite at any given moment, but capable of being infinitely extended whenever the machine comes to an end of the finite portion (an important assumption in view of the planned hardware implementation).

The head is situated, at any given moment, on a single square of the tape and has to carry out three operations to complete one step of the computation (one operation cycle of the finite-state machine). These operations are:

1. reading the symbol stored in the accessed location on the tape;
2. writing a symbol in the accessed location, erasing the previous symbol (of course, the latter can be preserved if the machine writes the same symbol that was read);
3. displacing the head left or right to an adjacent location (which becomes the accessed location for the next computation step).

A Turing machine can therefore be described by three functions f_1, f_2, f_3:

$$Q+ = f_1(Q, S)$$
$$S+ = f_2(Q, S)$$
$$D+ = f_3(Q, S)$$

where Q and S are, respectively, the current internal state of the finite state machine (FSM) and the current input symbol (the symbol in the accessed location on the tape), and where $Q+$, $S+$, and $D+$ are, respectively, the next internal state of the FSM, the symbol to be written in the accessed location, and the direction (left or right) of the head's displacement [12].

As a consequence, a set of *quintuples* can be used to specify what the machine will do for each possible combination of symbol and state. These quintuples have the following form:
(*current state, current symbol, next state, next symbol, direction of motion*)

or, equivalently:

$$(Q,\ S,\ Q+,\ S+,\ D+)$$

where the third, fourth, and fifth symbols are determined by the first and second according to the three functions f_1, f_2, and f_3 mentioned above.

These quintuples indicate that if a Turing machine is currently in the internal state Q, and if the current input symbol is S, the machine will change its internal state to the state $Q+$, replace the input symbol on the tape by the symbol $S+$, and move the read/write head by one location in the direction $D+$. If a Turing machine is in a condition for which it has no instruction, it halts.

The information contained in the set of quintuples is often represented in the form of a state table, defining the behavior of the machine for each possible combination of symbol and state.

An important observation, in view of the definition of the universal Turing machine that will follow in the next subsection, is that, because the head can move either way along the tape, it is possible for it to return to a previously printed location to recover the information inscribed there. This ability provides the machine a sort of rudimentary memory in a sense that the machine can look up the previous symbols and change them if necessary. Since the tape is as long as desired, this memory is potentially infinite.

2.2   Universal Turing Machines

Turing had the idea of the universal Turing machine (UTM), capable of simulating the operation of any specialized Turing machine, and gave an exact description of such a UTM in his paper [21]. The importance of the universal Turing machine is clear. We do not need to have an infinity of different machines doing different jobs. A single one will suffice [2].

A universal Turing machine, U, is a Turing machine with the property of being able to read the description (on its tape) of any other Turing machine, T, and to behave as T would have. The machine U consists of a finite-state machine (the program of U) controlling a mobile head, which operates on a tape. The data on the tape completely describe the machine T to be simulated (the data of T and the program of T, i.e., the three functions $Q+$, $S+$, and $D+$ describing T).

Figure 2 shows the organization of U's tape. To the left is a semi-infinite region containing the data of T's tape. Somewhere in this region is a marker M indicating where T's head is currently located. The middle region contains the current internal state Q and the current input symbol S of T. The right-hand region is used to record the description of T, i.e., the three functions $Q+$, $S+$, and $D+$ for each combination of Q and S.

The subject of this contribution is the realization of a universal Turing machine in hardware. The requirements of digital electronics in general and of the Embryonics architectures in particular have had an impact on the implementation choices for our UTM. For example, while theoretically a single
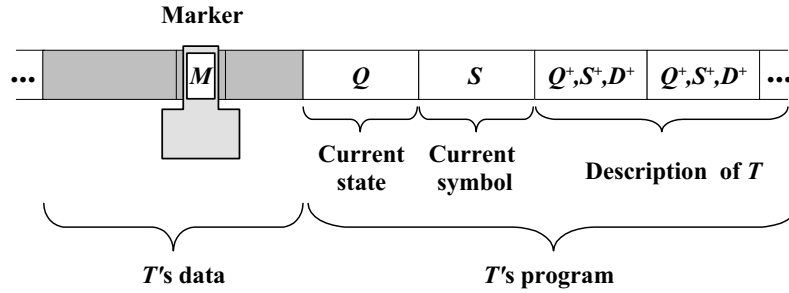
Fig. 2. Universal Turing machine's tape, describing the specialized machine T

tape is sufficient to store both the data and the program, an alternative but equivalent architecture separates the *program tape* from the data tape. It is this latter architecture, better suited to our Embryonics machines, that we adopted for our implementation.

3   Self-Replication of a Universal Turing Machine on a Multicellular Array

3.1   Embryonics Architectures

Living organisms are complex systems exhibiting a range of desirable characteristics, such as evolution, adaptation, and fault tolerance, that have proved difficult to realize using traditional engineering methodologies. The last three decades of investigations in the field of molecular biology (embryology, genetics, and immunology) has brought a clearer understanding of how living systems grow and develop. The principles used by Nature to build and maintain complex living systems are now available for the engineer to draw inspiration from [10].

The growth and the operation of all living beings are directed through the interpretation, in each of their cells, of a chemical program, the DNA. This program, called *genome*, is the blueprint of the organism and consists of a complex sequence written with an alphabet of four characters: A, C, G, and T. This process is the source of inspiration for the *Embryonics* (*embryonic electronics*) project [6,16,5,9], whose final objective is the conception of very large scale integrated circuits endowed with properties usually associated with the living world: self-repair and self-replication.

The MICTREE (for *tree of micro-instructions*) cell is a new kind of *coarse-grained field-programmable gate array* (*FPGA*), developed in the framework of the Embryonics project, which will be used for the implementation of multicellular artificial organisms with biological-like properties, i.e., capable of self-repair and self-replication [7,17].
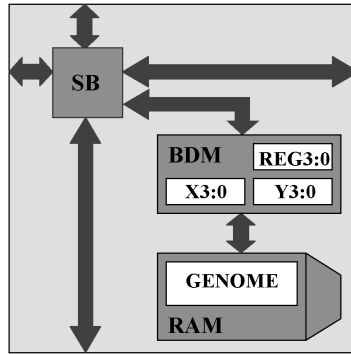
Fig. 3. MICTREE block diagram; SB: switch block; BDM: binary decision machine; RAM: random access memory; *REG3:0* : state register; *X3:0* : horizontal coordinate; *Y3:0* : vertical coordinate

MICTREE is a truly *cellular automaton* and its conception derives from the study of the multicellular living beings. It relies on three fundamental features: multicellular organization (the artificial organism is decomposed into a finite number of cells, where each cell realizes a unique function, described by a sub-program called the gene of the cell), cellular differentiation (the behavior of the cell depends on the physical position of the cell in the two-dimensional space, i.e., on its coordinates), and cellular division (starting from a mother cell, storing the one and only copy of the genome, a new cell can be programmed to store an exact copy of the genome).

The environment in which our quasi-biological artificial cells will develop consists of a finite (but as large as desired) two-dimensional space of silicon. This space is divided into rows and columns whose intersections define the cells. Since such cells (small processors and their memory) have an identical physical structure, i.e., an identical set of logic operators and of connections, the cellular array is homogeneous. Only the state of the cell, that is, the content of its registers, can differentiate it from its neighbors.

In all living beings, the string of characters which makes up the DNA, the *genome*, is executed sequentially by a chemical processor, the *ribosome*. Drawing inspiration from this mechanism, MICTREE is based on a *binary decision machine* (BDM) [4] (our ribosome), which sequentially executes a microprogram (our genome). In addition, the artificial cell is composed of a random access memory (RAM), and a communication system implemented by a switch block (SB) (Figure 3).

The *binary decision machine* executes a microprogram of up to 1024 instructions (the format of these instructions will be detailed later), which is stored in the RAM. The microprogram itself is decomposed in sub-programs that are equivalent to the different parts of the genome: the genes. The ex-
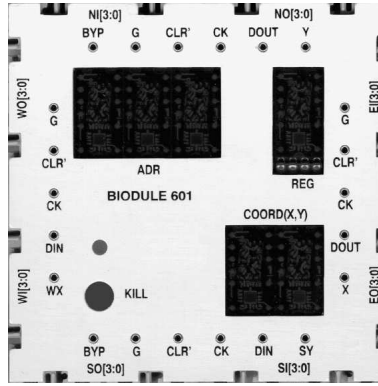
Fig. 4. BIODULE 601 demonstration module

ecution of a specific gene depends on the physical position of the cell in the two-dimensional array, i.e., on its coordinates.

As in nature, the entire microprogram of the organism is stored in each cell. This redundancy enormously simplifies the implementation of the desired properties of self-repair and self-replication:

- A set of BIST (Built-In Self-Test) techniques [1,11,19,18] detects the presence of faults within a cell. The column of cells containing the faulty cell is deactivated, and its functionality taken up by the column to its right, whose functionality is itself shifted to the right, and so on until a *spare column* is reached. The presence of the entire genome in each cell implies that this self-repair mechanism needs only a re-computation of the coordinates, without complex data transfers.
- The self-replication of an artificial organism rests on two hypotheses: (1) there exists a sufficient number of spare cells (i.e., the array is sufficiently large to hold more than one copy of the organism) and (2) the calculation of the coordinates produces a cycle. If these two hypotheses are met, the computation of the coordinates in the array automatically creates multiple copies of the artificial organism.

The MICTREE cell has been embedded into a plastic demonstration module, the BIODULE 601 (Figure 4). Each module can be easily joined with others, like a LEGO, to build larger artificial organisms. The size of the artificial organism embedded in an array of MICTREE cells is limited in the first place by the coordinate space ($X = 0...15$, $Y = 0...15$), that is, a maximum of 256 cells for the BIODULE 601 implementation), and then by the size of the memory of the binary decision machine storing the genome microprogram (1024 instructions).
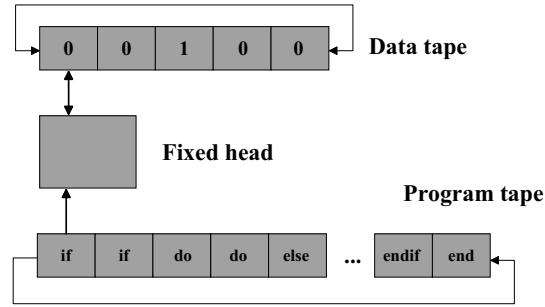
Fig. 5. Universal Turing machine architecture

3.2   Multicellular Architecture of a Universal Turing Machine

As we have seen, conventional universal Turing machines [12] consist of a finite but arbitrarily long tape, and a single read/write mobile head controlled by a finite-state machine, which is itself described on the tape (Figure 2). In order to implement a universal Turing machine in an array of MICTREE artificial cells, we made three fundamental architectural choices (Figure 5):

1. The read/write head is fixed; the tapes are mobile.
2. The data of the given application (the specialized Turing machine to be simulated) are placed on a mobile tape, the *data tape*; this tape can shift right, shift left, or stay in place.
3. The finite-state machine for the given application is translated into a very simple program written in a language called PICOPASCAL (Section 4); each instruction of this program is placed in a square of a second mobile tape, the *program tape*; this tape just needs to shift left. The transformation of a state table into such a program is directly inspired by the W-machine [23] with the major contribution of avoiding the jumps required by the **if** 1 **then** (*n*) **else** (*next*) instructions.

The fixed head, which is in fact an interpreter of the PICOPASCAL language, has to continuously execute cycles consisting of four operations:

1. reading and decoding an instruction on the program tape;
2. reading a symbol on the data tape;
3. interpreting the current instruction, and writing a new symbol on the data tape;
4. shifting the data tape (left, or right, or not at all) and the program tape (left).

3.3   An Application: a Binary Counter

In order to test our UTM implementation (Figure 5), we used, as a simple but non-trivial example, a binary counter [12], a machine that writes out the

binary numbers 1, 10, 11, 100, etc., the size of the numbers being limited only by the dimensions of the data tape. The counter's state table (Figure 6) has two internal states ($Q \in \{0 \rightarrow, 1 \leftarrow\}$) and two input states ($S \in \{0, 1\}$), S being the value of the current square read on the data tape. In this example we combined the internal state Q with the direction of the tape. For $Q = \{0 \rightarrow\}$ the data tape will move to the right, for $Q = \{1 \leftarrow\}$ the data tape will move to the left. Depending on the values of Q and S, the specialized Turing machine will:

1. write a new binary value $S+$ (0, 1) on the current square of the data tape;
2. move its data tape to the right ($Q+ = 0 \rightarrow$) or to the left ($Q+ = 1 \leftarrow$), which is equivalent to moving a mobile head to the left or to the right, respectively;
3. go to the next state $Q+$ (0 $\rightarrow$, 1 $\leftarrow$).

$Q+,S+$	$S=0$	$S=1$
$Q= 0\rightarrow$	$0\rightarrow,0$	$1\leftarrow,1$
$Q= 1\leftarrow$	$0\rightarrow,1$	$1\leftarrow,0$

Fig. 6. State table of the binary counter

The PICOPASCAL program equivalent to the state table (Figure 6) is given in Figure 7.

ADR	DATA	PROGRAM
00	5	if (Q)
01	5	if (S)
02	A	do 0 (S)
03	9	do 1← (Q)
04	4	else
05	B	do 1 (S)
06	8	do 0→(Q)
07	6	endif
08	4	else
09	5	if (S)
0A	B	do 1 (S)
0B	9	do 1← (Q)
0C	4	else
0D	A	do 0 (S)
0E	8	do 0→(Q)
0F	6	endif
10	6	endif
11	2	end

Fig. 7. PICOPASCAL program equivalent to the state table of Figure 6

3.4   An Ideal Architecture for the Universal Turing Machine

A universal Turing machine architecture, ideal in the sense that it is able to deal with applications of any complexity, is characterized by:

1. a finite, but arbitrarily long data tape;
2. a read/write head able to interpret a PICOPASCAL program of any complexity;
3. a finite, but arbitrarily long program tape.

It must be pointed out that, for any application, the program tape and the read/write head (the PICOPASCAL interpreter) are always characterized by finite and defined dimensions; only the data tape can be as long as desired, as is the case for the binary counter, whose growth is potentially infinite.

An ideal architecture, embedding the current example, but compatible with any other application, could be the following (Figure 8):

1. The data tape, able to shift right, to shift left, or hold, is folded on itself. The initial state is defined in Figure 8 by $QL3{:}0$, QC, $QR0{:}3 = 000010000$, where QL are the squares to the left of the central square QC, and QR are the squares to the right of QC; the data tape is able to grow to the left of QC, i.e., to the right of $QL3$ ($QL4,QL5$, ...) and to the right of QC ($QR4$, $QR5$, ...), as can be appreciated in Figure 8.
2. The fixed read/write head, which will be detailed in Section 5, is basically composed of a state register Q,S (storing the current values of internal and input states Q,S, respectively, with an initial state $Q,S = 01$) and a stack $ST1{:}3$ characterized by a 1-out-of-3 code (one-hot encoding). At the start of the execution of the PICOPASCAL program (i.e., in Figure 7, at address $ADR = 00$), the stack is in an initial state $ST1{:}3 = 100$. Roughly speaking, each **if** instruction will involve a PUSH operation, each **endif** a POP operation, and each **else** a LOAD operation. When $ST1 = 1$, the **do** instructions are executed. The main characteristic of the stack is its scalability: for any program exhibiting n nested **if** instructions, the stack is organized as a $n+1$ squares shift register. Both the $ST1{:}3$ stack and the Q,S register are able to grow to accommodate more complex applications.
3. The program tape is folded on itself; it is able to grow to accommodate more complex applications.

This ideal architecture is moreover compatible with the Embryonics concept: self-replication may be accomplished along the vertical axis, self-repair along the horizontal axis, and the scalable properties of both the data tape and the fixed head (stack and Q,S registers) are compatible with the limited number of distinct coordinates (a scalable and regular architecture may be described by repetition of the same type of cells, i.e., of the same coordinates).
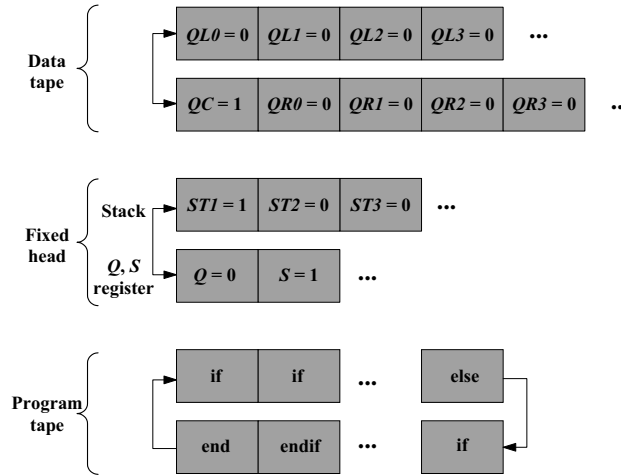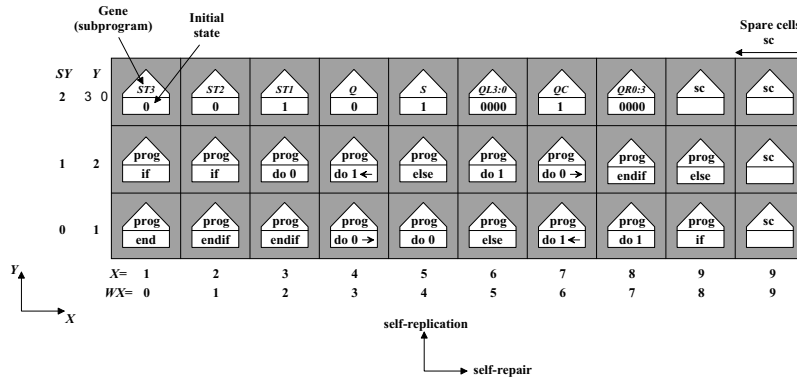
Fig. 8. UTM's ideal architecture

Fig. 9. UTM's actual implementation for the binary counter example on a multicellular array of 27 MICTREE cells plus 3 spare cells (sc). *WX*: horizontal coordinate of the western neighboring cell; *SY*: vertical coordinate of the southern neighboring cell

3.5 An Actual Implementation of the Universal Turing Machine

In order to implement the binary counter application with a limited number of MICTREE artificial cells, we have somewhat relaxed the requirements of the ideal architecture described earlier. Our final architecture is made up of three rows ($Y = 1...3$) and nine columns ($X = 1...9$) organized as follows (Figure 9):

- The 18 instructions of the PICOPASCAL program (Figure 7) are placed in the program tape, using the two lower rows ($Y = 1, 2$) of the array.
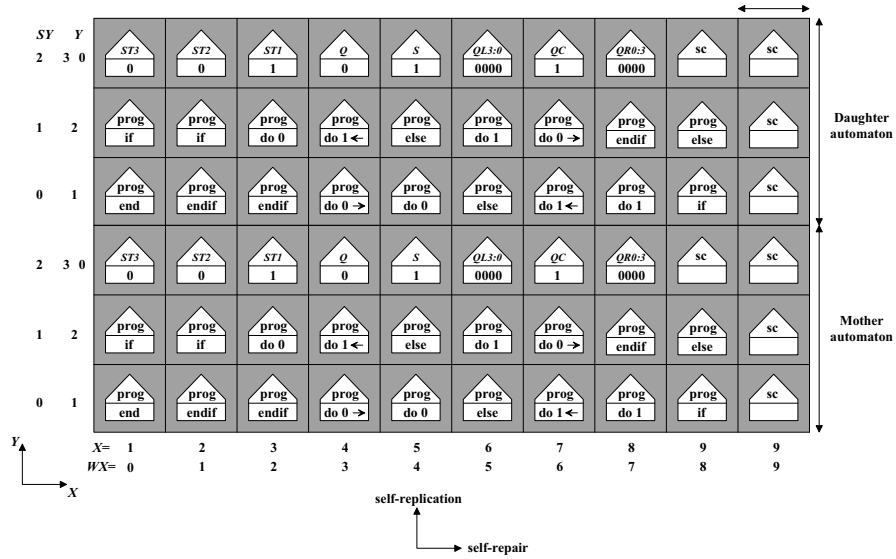
SY	Y	X=1 (WX=0)	2 (1)	3 (2)	4 (3)	5 (4)	6 (5)	7 (6)	8 (7)	9 (8)	9 (9)
2	3 0	ST3 0	ST2 0	ST1 1	Q 0	S 1	QL3:0 0000	QC 1	QR0:3 0000	sc	sc
1	2	prog if	prog if	prog do 0	prog do 1 ←	prog else	prog do 1	prog do 0 →	prog endif	prog else	sc
0	1	prog end	prog endif	prog endif	prog do 0 →	prog do 0	prog else	prog do 1 ←	prog do 1	prog if	sc
2	3 0	ST3 0	ST2 0	ST1 1	Q 0	S 1	QL3:0 0000	QC 1	QR0:3 0000	sc	sc
1	2	prog if	prog if	prog do 0	prog do 1 ←	prog else	prog do 1	prog do 0 →	prog endif	prog else	sc
0	1	prog end	prog endif	prog endif	prog do 0 →	prog do 0	prog else	prog do 1 ←	prog do 1	prog if	sc

Daughter automaton (upper three rows); Mother automaton (lower three rows). self-replication; self-repair.

Fig. 10. *Self-replication of the UTM's actual implementation for the binary counter example.* WX*: horizontal coordinate of the western neighboring cell;* SY*: vertical coordinate of the southern neighboring cell*

- The read/write head is composed of a *ST1:3* stack and of the *Q,S* register ($X = 1...5$, $Y = 3$), while the data tape is implemented by three cells ($X = 6...8$, $Y = 3$) storing 9 bits *QL3:0, QC, QR0:3*.

In order to demonstrate self-repair, we added spare cells to each row, at the right-hand side of the UTM, all identified by the same horizontal coordinate ($X = 9$ in Figure 9). As previously mentioned, more cells may be used not only for self-repair, but also for a UTM necessitating a growth of the tape of arbitrary, but finite, length.

Self-replication rests on two hypotheses (Figure 10):

- there exist a sufficient number of spare cells (unused cells at the upper side of the array, at least $3 \times 9 = 27$ for our example);
- the calculation of the coordinates produces a cycle at the cellular level (in our example: $Y = 1 \rightarrow 2 \rightarrow 0 \rightarrow 1 \rightarrow 2 \rightarrow 0$).

Given a sufficiently large space, the self-replication process can be repeated for any number of specimens in the Y axis. With a sufficient number of cells, it is obviously possible to combine self-repair (or growth) towards the X direction and self-replication towards the Y direction.

In the next section we will present the PICOPASCAL language and a PICOPASCAL interpreter architecture, which are necessary for the understanding of our Embryonics implementation.

4   PICOPASCAL

4.1   The PICOPASCAL Language

The PICOPASCAL language consists of a minimal subset of the MODULA-2 language [24]. PICOPASCAL is thus a *high-level* language: it does not make use of explicit addressing and provides a great simplicity of use. PICOPAS-CAL is, moreover, a *structured* language and thus guarantees, because of its structure, a rigorous and efficient notation. In conformity with this last feature, PICOPASCAL has three fundamental constructs, described below: (1) the sequence, (2) the choice or alternative, and (3) the iteration.

The assignment **do...**, realizing the synchronous transfer of a constant into a register, is a structured program. The *sequence* (or composition) of two such instructions *P1* and *P2*, written **do** *P1P2*, is a structured program, described by the flowchart and by the mnemonic program of Figure 11a. This last notation consists of a linear succession of instructions, displayed in the growing order of addresses *ADR*.

The *choice* (or alternative) of *P1* or *P2*, where *P1* and *P2* are two assignments, is a structured program, written **if** a **then** *P1* **else** *P2*. It is represented symbolically by the flowchart of Figure 11b, and realized by the linear succession of the instructions of the corresponding functional diagram and mnemonic program. To facilitate comprehension, and unlike programs written in a low-level language using explicit addresses, there is no jump (notably, to avoid the instruction *P1* when $a = 0$ or the instruction *P2* when $a = 1$): all instructions are read sequentially, from *ADR*=0 to *ADR*=4, and the execution of the assignments *P1* or *P2* depends on the value of a signal *EXEC* (for *EXECUTE*) which, in turn, depends on the value of the test variable a. This process will be revisited in detail in the description of the interpreter of the PICOPASCAL language (Subsection 4.2).

The last construct of structured programming, the *conditional iteration* **while** a **do** *P1*, is thus not necessary in the PICOPASCAL language. However, since our program must be continually executed, notably to allow self-repair, we allow the loop illustrated by the flowchart of Figure 11c, which in fact introduces a non-conditional iteration on the entire program.

In conclusion, the PICOPASCAL language is described by the *syntactic diagram* of Figure 11d, where we can count ten different terminal symbols (ovals), which make up the instructions of the language: **begin**, **end**, **NOP**, **do** 0, **do** 1, **do** 0→, **do** 1←, **if**, **else**, **endif**. The **NOP** (*No operation*) instruction represents the execution of a neutral operation.

Figure 12a shows the operating code (*OPC*) for the instructions of the PICOPASCAL language. Figure 12b and Figure 12c show the binary decision diagram of the binary counter example and its PICOPASCAL description, derived from the state table in Figure 6.
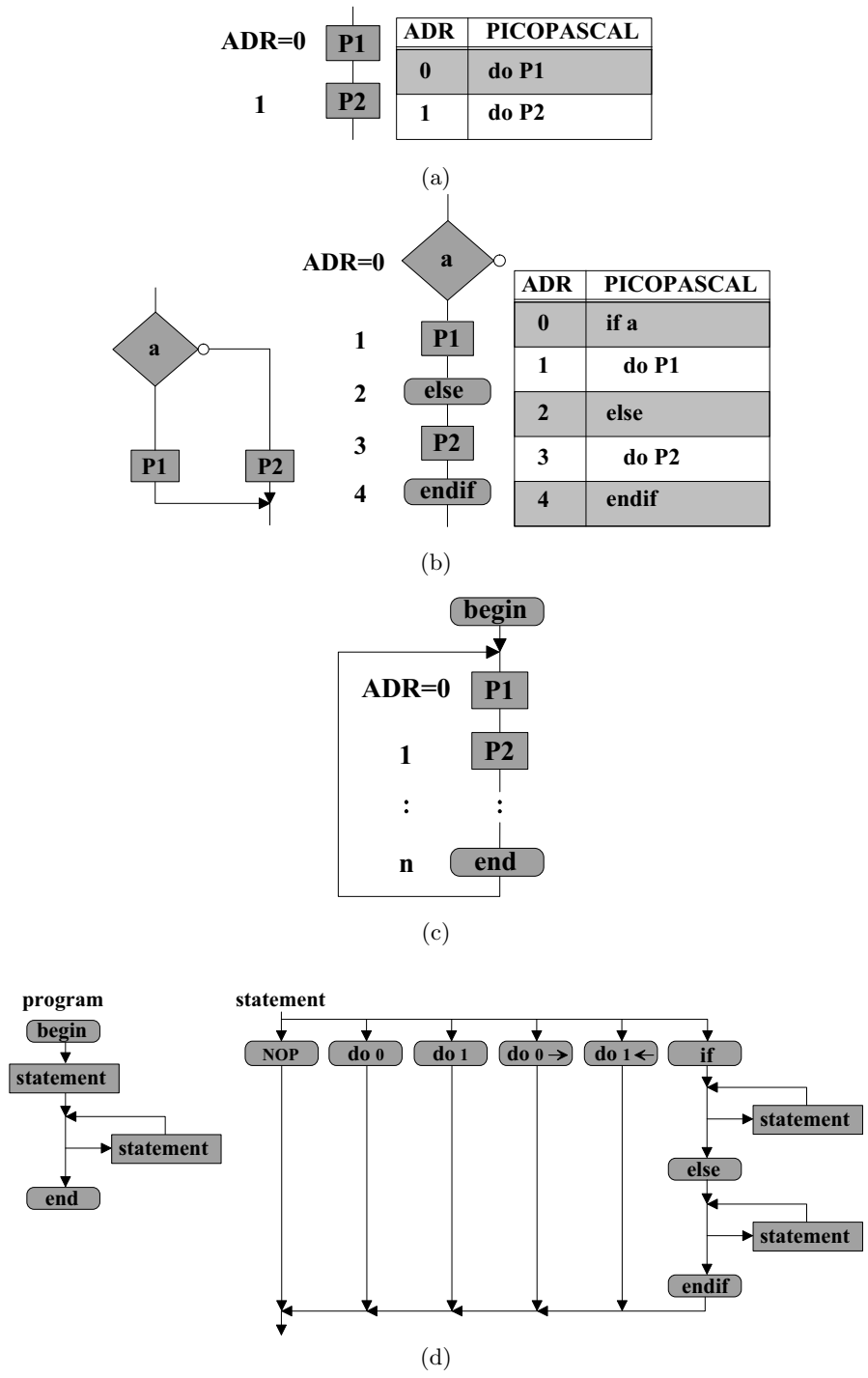
(a)

(b)

(c)

(d)

Fig. 11. *PICOPASCAL language. (a) Sequence of two assignment instructions:* **do** P1P2. *(b) Choice of either* P1 *or* P2: **if** *a* **then** P1 **else** P2. *(c) Non-conditional iteration loop. (d) Syntactic diagram*

OPC3:0	OPC	Instruction	OPC3:0	OPC	Instruction
0000	0	NOP	1000	8	do 0 →
0001	1	begin	1001	9	do 1 ←
0010	2	end	1010	A	do 0
0011	3		1011	B	do 1
0100	4	else	1100	C	
0101	5	if	1101	D	
0110	6	endif	1110	E	
0111	7		1111	F	

(a)
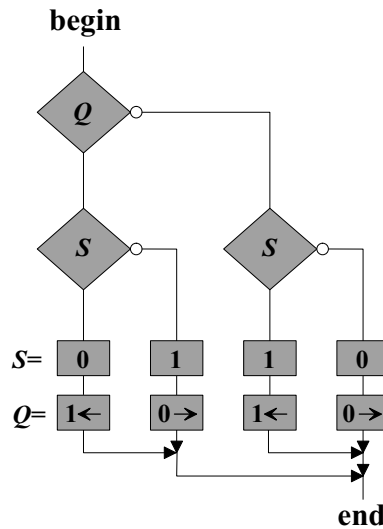

```
begin
if (Q)
    if (S)
        do 0
        do 1←
    else
        do 1
        do 0→
    endif
else
    if (S)
        do 1
        do 1←
    else
        do 0
        do 0→
    endif
endif
end
```

(b)   (c)

Fig. 12. *PICOPASCAL language. (a) Opcodes for the ten instructions of the PI-COPASCAL language. (b) Binary decision diagram of the binary counter example. (c) PICOPASCAL description*

4.2   PICOPASCALINE: an Interpreter for the PICOPASCAL Language

Figure 13 suggests a possible hardware architecture to execute the ten instructions of the PICOPASCAL language. From now on, we will refer to this machine as PICOPASCALINE. The instruction **end**, as well as the pseudo-instruction **begin** (not executed), have the same effect: jumping to the instruction at address 0 (note that in this architecture the **begin** instruction is not necessary and can thus be removed). The **if** instruction does not require a test variable, since the hardware is capable of presenting the correct variable at the right time. There exist therefore nine distinct types of instruction to interpret.
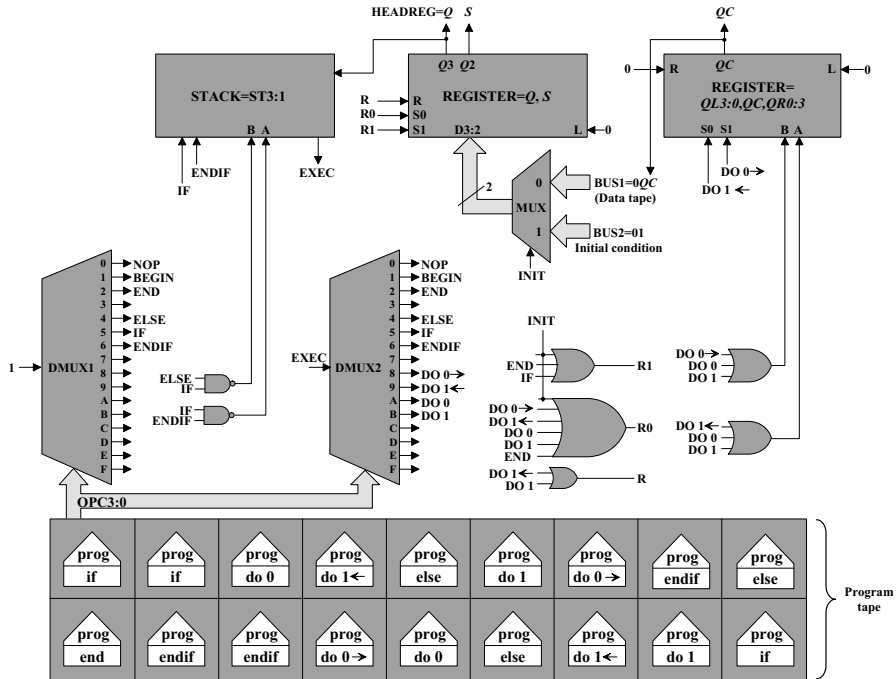
Fig. 13. PICOPASCALINE: PICOPASCAL interpreter for the ten instructions of the language

To decode the instructions (*OPC3:0*) on the program tape, the PICO-PASCALINE consists of the following elements (Figure 13):

- A state register REGISTER storing the current values of the internal and input states Q, and S respectively, with an initial state $Q,S = 01$.
- A register REGISTER storing the values $QL3:0,QC,QR0:3$ of the data tape, with an initial state $QL3:0,QC,QR0:3 = 000010000$.
- A stack STACK characterized by a 1-out-of-3 code (one-hot encoding), with an initial state STACK = $ST3:1 = 001$.
- A decoder DMUX1 controlled by the 4 bits of the operating code $OPC3:0$, which generates the signals controlling the STACK (signals *IF*, *ELSE*, and *ENDIF*).
- A decoder DMUX2 controlled by the 4 bits of the operating code $OPC3:0$ and by the *EXEC* signal. This decoder generates the signals controlling the (Q,S) and $(QL3:0,QC,QR0:3)$ REGISTERs (signals *DO* 0, *DO* 1, *DO* 0→, *DO* 1←, *IF*, and *ELSE*).
- A multiplexer MUX controlled by the signal *INIT*, which selects one of the two input busses, *BUS1* coming from the data tape, or *BUS2* which is a constant used for initialization purposes. At the start of the execution

the signal *INIT* has the value 1 and the (*Q*,*S*) REGISTER is initialized, whereas the rest of the execution this variable takes the value 0 and the value *QC* coming from the (*QL3:0,QC,QR0:3*) REGISTER is assigned to the (*Q*,*S*) REGISTER.

The signal *EXEC* controls the execution of the assignment instructions **do** and thus depends on the succession of values of the internal and input states *Q* and *S*. We will now examine this process for the example of the program of Figures 7 and 12, whose detailed execution is shown in Figure 14. We assume that the values of the test variables are *Q*=1 and *S*=0, and that these values do not change during the execution of the microprogram. Disposing of a stack (STACK) of 1-bit wide and three levels deep, we observe the following chronology:

- At the start of the program's execution, the three levels of the stack are initialized to the value *ST3:1*=001. The signal *EXEC*, which is the value at the top of the STACK, i.e., *ST1*, is thus equal to 1.
- The first logic test (**if** *Q*) produces a value 1 which is placed at the top of the stack (operation PUSH). *EXEC* keeps the value 1.
- The second test (**if** *S*) produces a value 0 which in turn is placed at the top of the stack (PUSH). *EXEC* is reset to 0.
- Since the *EXEC* signal is 0, the assignment **do** 0, and **do** 1← are not executed (NOP): the stack remains in a neutral state (NOP operation) and the *EXEC* signal is still 0.
- The instruction **else** indicates the passage from the left branch of the test (**if** *S* **then** *P1*) to the right branch (**else** *P2*). It corresponds to a COMPLEMENT operation, where the top of the STACK (*ST1 = EXEC*) is inverted, while the content is changed to maintain the 1-out-of-3 code. The signal *EXEC* is again set to 1.
- Since *EXEC* is now 1, the assignments **do** 1, and **do** 0→ are executed.
- The instruction **endif** controls the popping of the stack (POP operation). The signal *EXEC* keeps the value 1 to maintain the 1-out-of-3 code.
- The execution of the program then continues as above until the final instruction **end**, where the stack finds again its initial state, with its first level in the state 1 (*EXEC*=1).

The operation table of Figure 15a describes the global operation of the stack, while the logic diagram of Figure 15b describes a possible realization of the stack, according to the table of operations of Figure 15a. With the exception of the first and second levels, we note the iterative nature of this stack, which contains six levels in this implementation and is thus capable of successively testing up to six variables (six is therefore the highest number of nested tests in this implementation).

The intrinsic limitations of the PICOPASCALINE interpreter are determined by the number and size of the registers, as well as by the number of
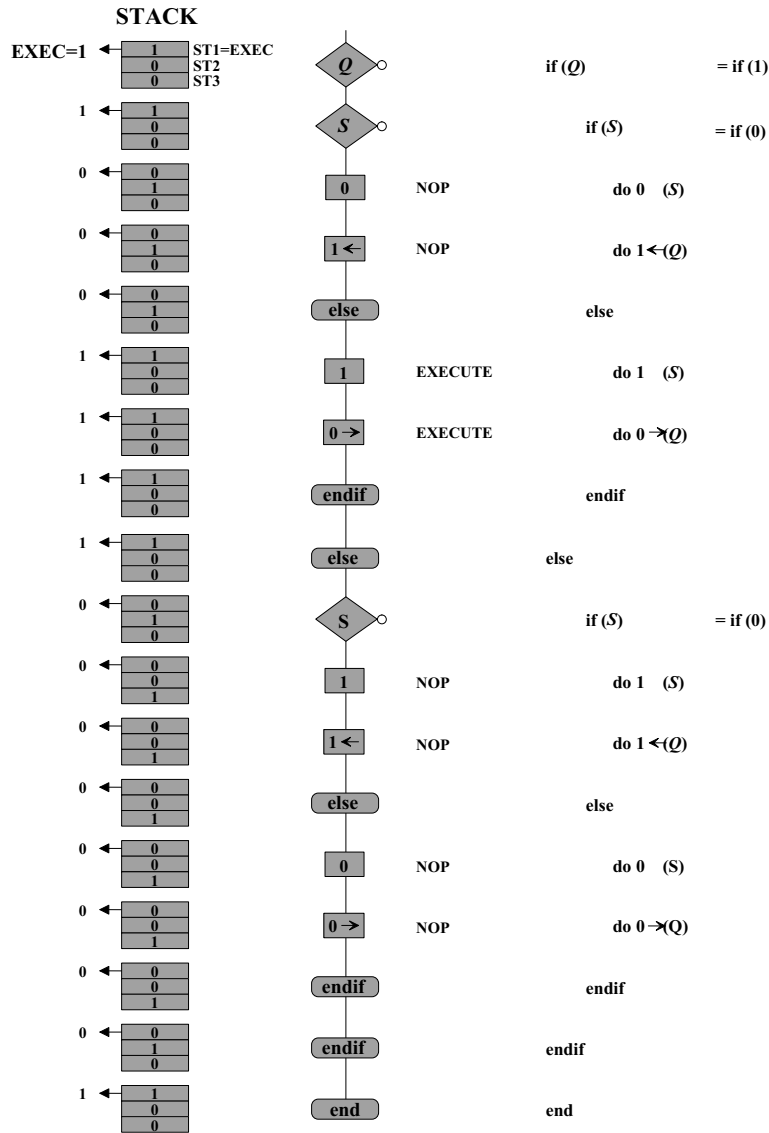
STACK

STACK	Flow		Code	
EXEC=1 ← 1 / 0 / 0 ST1=EXEC ST2 ST3	Q ◯		if (Q)	= if (1)
1 ← 1 / 0 / 0	S ◯		if (S)	= if (0)
0 ← 0 / 1 / 0	0	NOP	do 0 (S)	
0 ← 0 / 1 / 0	1 ←	NOP	do 1 ← (Q)	
0 ← 0 / 1 / 0	else		else	
1 ← 1 / 0 / 0	1	EXECUTE	do 1 (S)	
1 ← 1 / 0 / 0	0 →	EXECUTE	do 0 → (Q)	
1 ← 1 / 0 / 0	endif		endif	
1 ← 1 / 0 / 0	else		else	
0 ← 0 / 1 / 0	S ◯		if (S)	= if (0)
0 ← 0 / 0 / 1	1	NOP	do 1 (S)	
0 ← 0 / 0 / 1	1 ←	NOP	do 1 ← (Q)	
0 ← 0 / 0 / 1	else		else	
0 ← 0 / 0 / 1	0	NOP	do 0 (S)	
0 ← 0 / 0 / 1	0 →	NOP	do 0 → (Q)	
0 ← 0 / 0 / 1	endif		endif	
0 ← 0 / 1 / 0	endif		endif	
1 ← 1 / 0 / 0	end		end	

Fig. 14. *Interpretation of the program of Figures 7 and 12. The values of the test variables are Q=1 and S=0 (these values do not change during the execution of the PICOPASCAL program)*

tested variables which can be stored in the stack. A detailed description of the operation of the stack can be found in [13].
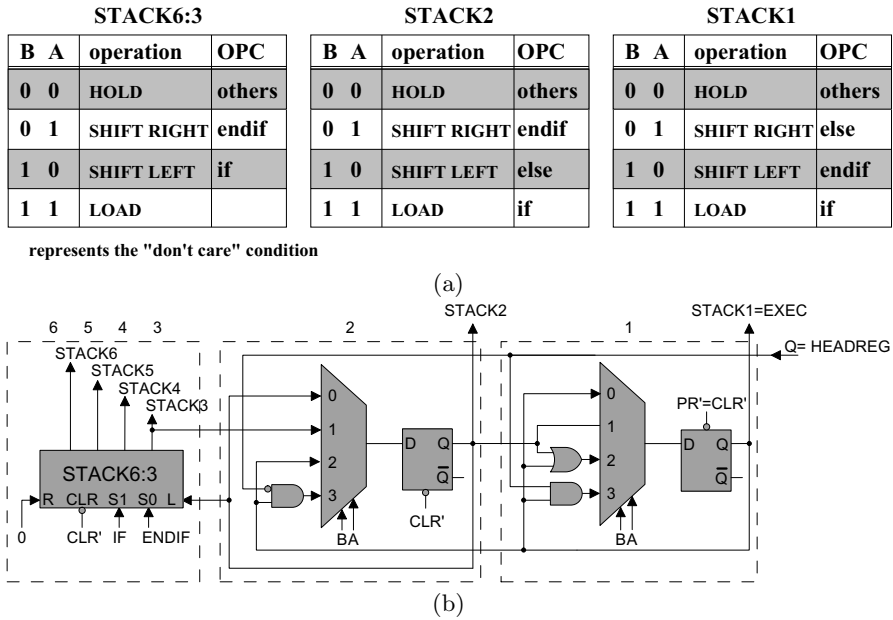
STACK6:3			
B A	**operation**	**OPC**	
0 0	HOLD	others	
0 1	SHIFT RIGHT	endif	
1 0	SHIFT LEFT	if	
1 1	LOAD		

STACK2			
B A	**operation**	**OPC**	
0 0	HOLD	others	
0 1	SHIFT RIGHT	endif	
1 0	SHIFT LEFT	else	
1 1	LOAD	if	

STACK1			
B A	**operation**	**OPC**	
0 0	HOLD	others	
0 1	SHIFT RIGHT	else	
1 0	SHIFT LEFT	endif	
1 1	LOAD	if	

represents the "don't care" condition

(a)

(b)

Fig. 15. 6-level PICOPASCALINE stack. (a) Operation tables. (b) Detailed architecture

The PICOPASCAL language and its interpreter were used to realize the architecture described in Section 3 and thus implement our self-replicating universal Turing machine. In the next section we will describe in some more detail the structure and operation of the genome microprogram of our organism.

5   Detailed Implementation of a Universal Turing Machine

5.1   The Genome Microprogram

As shown in Figure 9, the actual implementation of our UTM architecture consists of 27 cells, where each cell contains the entire genome of the organism and, depending on its position in the array, can interpret the genome and extract and execute the gene which configures it.

The genome microprogram thus consists of three main parts, as shown in the flowchart **UTMgenome** of Figure 16: first, the initial conditions for the machine are set, then the coordinates are computed (left loop: this process requires several iterations to allow for the propagation of the coordinates through the array), and finally the operative genome is executed (right loop). The latter can itself be decomposed into the distinct genes required by our

artificial organism (Figure 9): the *program tape genes*, the *stack genes*, the *register genes*, and the *data tape genes*.
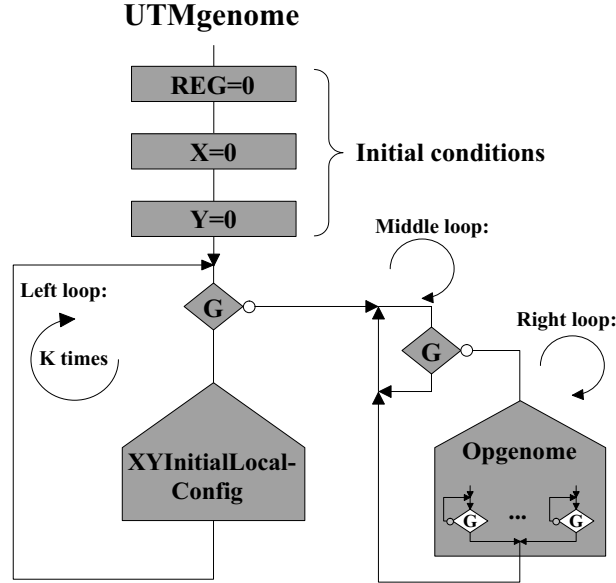

Fig. 16. Complete genome microprogram *(**UTMgenome**)* flowchart

The definition of the initial conditions assures that:

- all the state registers of the array are set to 0 ($REG = 0$);
- the coordinates X and Y are set to 0 ($X = 0$, $Y = 0$).

The microprogram then executes one of three loops, controlled by the variable G (the global clock):

1. For the first period of G, while $G = 1$, at least K executions of the left-hand loop are necessary to ensure that the rightmost and the uppermost cells of the organism correctly compute their coordinates, their initial conditions, and their local configurations (as defined by the sub-program **XYInitialLocalConfig**). K is the largest of $Xmax$ and $Ymax$ ($Xmax$ and $Ymax$ are, respectively, the maximum number of rows and columns of the artificial organism). In our example, $K = 10$ (Figure 10).
2. At the falling edge of G ($G = 1 \rightarrow 0$), the right loop is selected and the operational part of the genome is executed. To assure the synchronization of all the cells, tests are performed throughout the half-period when $G = 0$, but no assignment is made until the rising edge of G ($G = 0 \rightarrow 1$), when all the registers REG are updated simultaneously.

3. At the rising edge of G ($G = 0 \rightarrow 1$), after the update of the registers, the middle loop is executed. The program stays in this loop until the next falling edge of G ($G = 1 \rightarrow 0$) arrives, selecting the right-hand loop and starting a new cycle.

The following subsections will describe in detail the calculation of the operative genome in general, and of the program tape genes in particular. A detailed decription of the calculation of the other components of the genome (e.g., the computation of the coordinates and the definition of the initial conditions) can be found in [13].

5.2   Computing the Operative Genome

From the description of Figure 9, we can observe that our artificial organism is composed of four main parts:

- The *program tape* realizes the PICOPASCAL program tape and is implemented by the two lower rows ($Y = 1, 2$) of the array. Its architecture consists of a shift register, based on three different kinds of cells.
- The *stack* is implemented by three cells *ST3:1* ($X = 1..3$, $Y = 3$). Each of these three cells is different and is described by a specific gene.
- The *register* is implemented by two different cells *Q,S* ($X = 4, 5$, $Y = 3$) and is therefore described by two specific genes.
- The *data tape* is implemented by three different cells *QL3:0*, *QC*, *QR0:3* ($X = 6..8$, $Y = 3$) and is described by three specific genes.

Cellular differentiation (i.e., the definition of which part, or *gene*, of the complete genome will be executed in each cell) occurs through the vertical coordinate, computed as a function of the coordinate SY of the preceding cell (the southern neighbor), and through the horizontal coordinate, computed as a function of the coordinate WX of the preceding cell (the western neighbor). From Figure 9, we can show that the vertical coordinate SY can be used to differentiate the stack genes, the register genes, and the data tape genes (SY=2) from the program tape genes (SY=1,0).

The specifications of Figure 9 allow us to derive directly the Karnaugh map of Figure 19a, which defines the placement of the stack (*ST3:1*), register (*Q,S*), and data tape (*QL3:0*, *QC*, *QR0:3*) genes into the cellular space. The sub-tree contained by the leftmost dashed square of Figure 19b implements this Karnaugh map. To find the different genes of the program tape, we need to analyze in more detail its particular architecture.

Each cell of our program tape ($Y = 1, 2$) implements one PICOPASCAL instruction (stored in the four-bit *REG* register) and at each program step every instruction has to be shifted anticlockwise. From Figure 17, which shows the routing path established between each cell to transfer the instruction to its neighbor, we have to consider three different situations, which will be used to identify the position of the three specific genes:
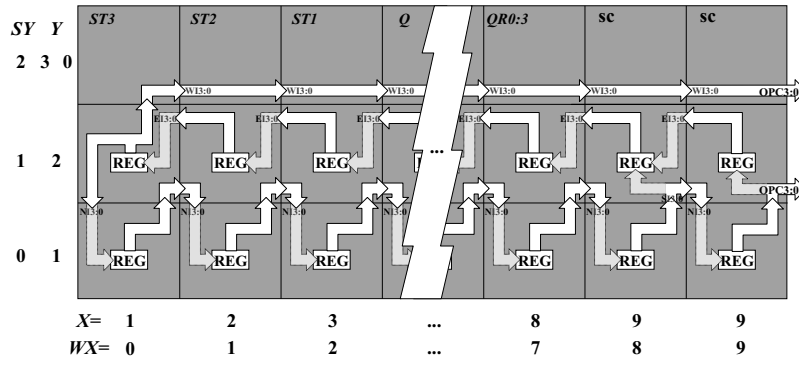
Fig. 17. Routing path established between the cells of the program tape ($SY = 0,1$) to implement the anticlockwise shift of the PICOPASCAL program
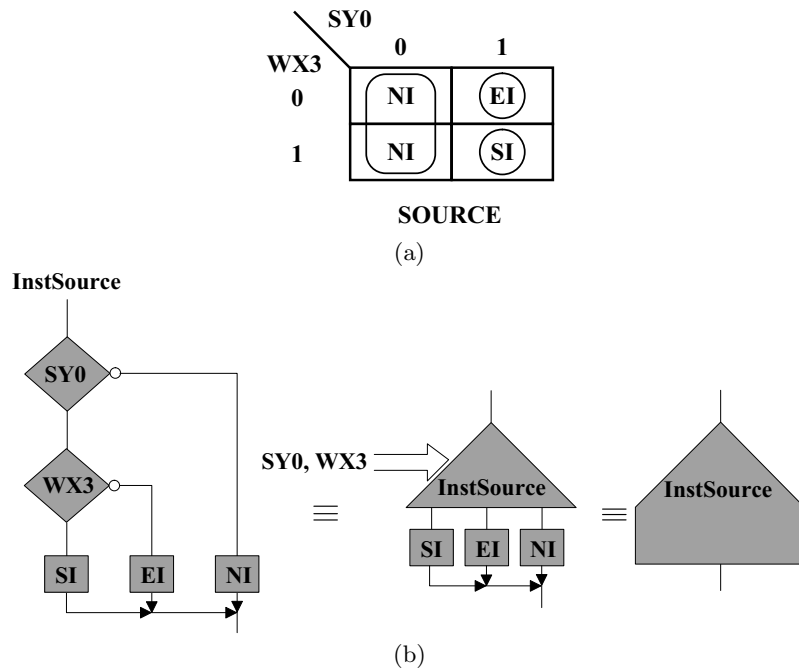

Fig. 18. InstSource program for the differentiation of the tape genes. (a) Karnaugh map. (b) Binary decision diagram and flowcharts

- the cells at coordinates ($WX = 0..7$, $SY = 1$) will receive, decode and store (in the REG register) the instruction of the east neighbor through the input bus $EI3:0$;

- the cells at coordinates ($WX = 8,9$, $SY = 1$) will receive, decode and store (in the REG register) the instruction of the south neighbor through the input bus $SI3{:}0$;
- the cells at coordinates ($WX = 0..9$, $SY = 0$) will receive, decode and store (in the REG register) the instruction of the north neighbor through the input bus $NI3{:}0$.

The functionality of each group of cells of the program tape can be expressed as a function of the horizontal coordinate ($WX3$) and of the vertical coordinate ($SY0$). We therefore have to solve a problem of two variables, as shown by Karnaugh map of Figure 18a. Figure 18b shows the resulting binary decision diagram.

By joining the binary decision diagram of Figure 18b (**InstSource**) and the binary decision diagram derived from the Karnaugh map of Figure 19a (sub-tree contained by the leftmost dashed square of Figure 19b), we can generate the binary decision diagram and flowchart of Figure 19b describing the complete operational part of the genome of our UTM implementation (**Opgenome**).

5.3   Computing the Program Tape Genes

Figure 9 ($Y = 1, 2$) shows that the program tape, in our implementation, is composed of 18 cells and features three different genes (EI, SI, and NI). Since the **begin** instruction is not executed and the **NOP** instruction is not used, these cells have to deal with a program composed of eight different kinds of PICOPASCAL instructions (Figure 12a).

Therefore, each cell has to decode the instruction coming from its neighbor (east, south, or north) and store it in the REG register. The Karnaugh map of Figure 20a shows the binary coding proposed for the eight PICOPASCAL instructions stored in the program tape. Figure 20b shows the generic binary decision diagram that we use to decode the instruction to be assigned to the register REG. In consequence, we can implement the three genes EI, SI, and NI, of the program tape by decoding the instructions coming from the east, south, and north neighbors respectively, that is, by replacing $OPC3{:}0$ (Figure 20b) by $EI3{:}0$, $SI3{:}0$, and $NI3{:}0$ respectively.

To assure the synchronization of all the registers, tests are performed throughout the half-period when $G = 0$, but no assignment is made until the rising edge of G ($G = 0 \rightarrow 1$), when all the registers REG are updated simultaneously.

As shown in Figure 9, the stack part of our cellular UTM implementation is composed of the three cells $ST3{:}1$ ($X = 1..3$, $Y = 3$), each featuring a different gene ($ST1$, $ST2$, and $ST3$). The Embryonics implementation of the $STACK$ part of our artificial organism (the PICOPASCALINE stack) has to reproduce the behavior described by the stack operation tables in Figure 15a and by the stack architecture presented in Figure 15b. From the
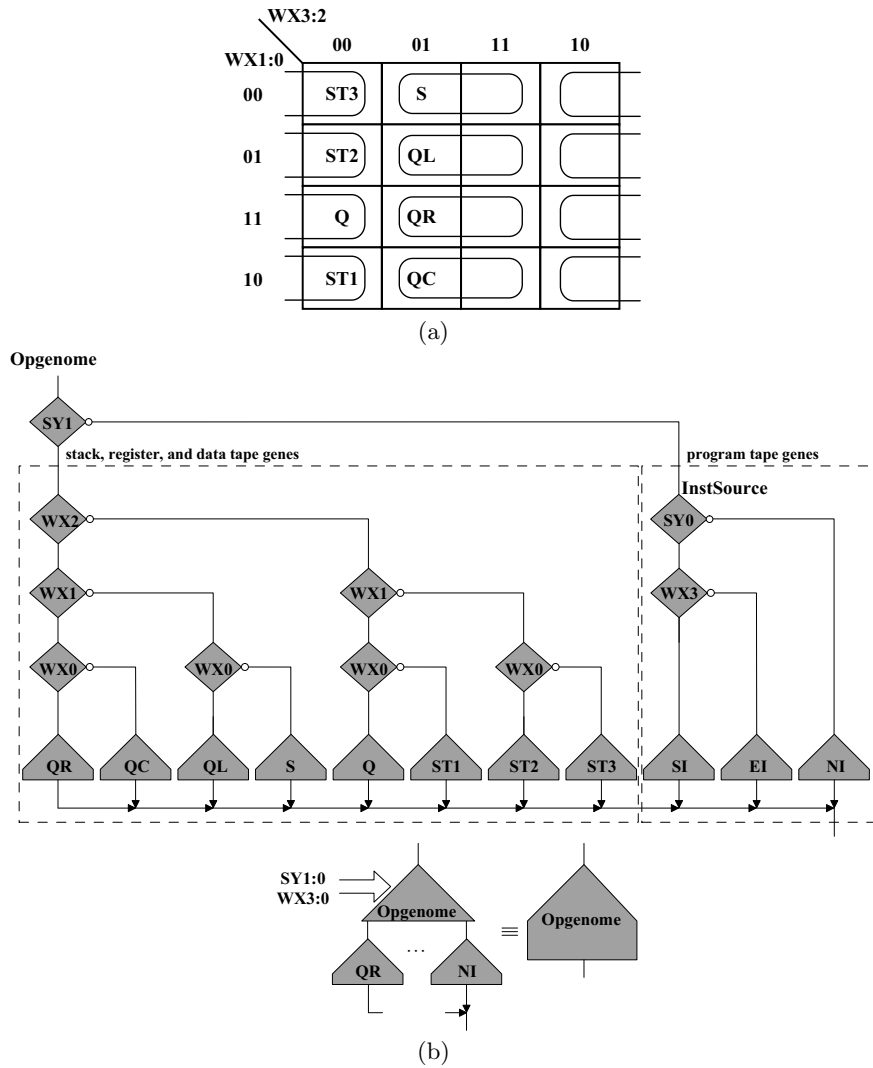
(a)

(b)

Fig. 19. Computing the genome's operational part (sub-program **Opgenome**). (a) Karnaugh map for stack (ST3:1), register (Q,S), and data tape genes (QL3:0, QC, QR0:3). (b) Binary decision diagram and flowchart of the genome's operational part

tables we obtain the information to build the Karnaugh maps decoding the PICOPASCAL instructions related with each stack gene (*ST1, ST2, ST3*), and from the architecture we obtain the logic part and the corresponding control signals.
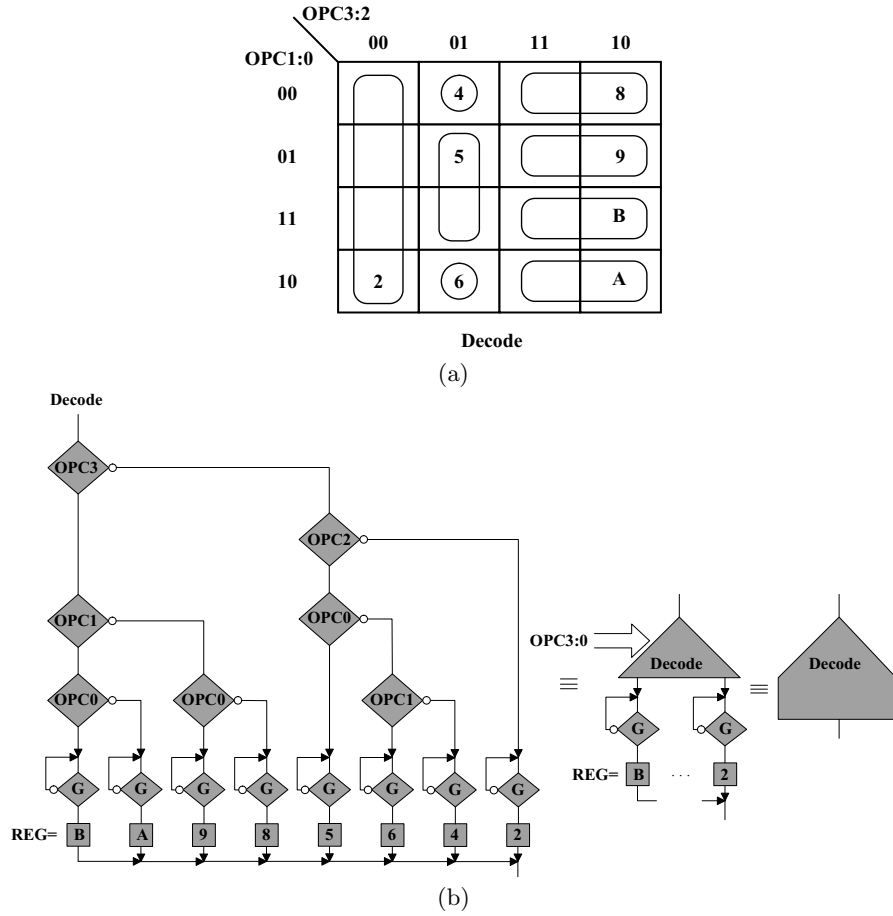
(a)

(b)

Fig. 20. *Generic decoding of the instruction to be assigned to REG. (a) Karnaugh map. (b) General binary decision diagram and flowcharts* (**Decode**)

6   Conclusion

In this contribution we showed that it is possible to embed a universal Turing machine into a multicellular array based on MICTREE artificial cells, thus obtaining a self-repairing and self-replicating universal Turing machine.

The mapping of the universal Turing machine onto our multicellular array was made possible thanks to the introduction of a modified version of the W-machine [23], i.e., an interpreter of the PICOPASCAL language. We showed that an ideal architecture (i.e., an architecture with a semi-infinite data tape) was able to deal with applications of any complexity. We also presented an actual implementation in which we relaxed somewhat the requirements of the ideal architecture in order to use a smaller number of our MICTREE artificial
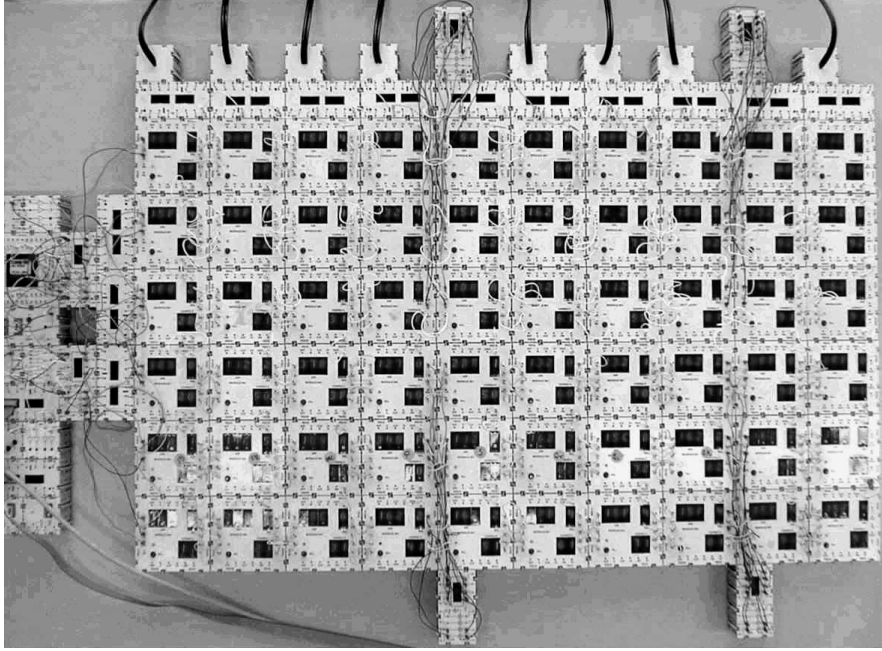
Fig. 21. Final universal Turing machine implementation. This implementation contains six rows and ten columns of MICTREE cells, allowing us to verify self-replication (one copy) and self-repair (one spare column)

cells. We slightly simplified our implementation by presenting the example of the binary counter in which the data are binary-coded and where the direction of the head's motion coincides with the internal state (in general, functions $Q+$ and $D+$ are independent). A picture of the final implementation is shown in Figure 21.

The UTM was completely implemented and the binary counter fully tested. The values obtained correspond exactly to the results presented by Minsky in [12], assuming that the tape register is limited to 9 bits with fixed boundary conditions ($R=L=0$ in Figure 13). The measured sequence ends in a final "quiescent" state (where all the symbols on the data tape are 0's), which constitutes a fixed point for the system.

The complete genome microprogram describing our artificial organism is composed of 377 16-bit-wide instructions, implying a configuration bit string of 6032 bits.

We tested the self-repair capabilities of our implementation (Figure 9), made possible by the spare column at the right edge of our artificial organism. Using this spare column, our organism is able to tolerate at least one fault in any cell of the array, and up to three faulty cells in the same column.

The self-replication of our UTM was tested with one copy of the original organism, as shown Figure 10. For this test, the cellular array contained 6x10 = 60 MICTREE cells (Figure 21).

The property of universal construction, another challenge laid down by von Neumann's original self-replicating automaton, raises issues of a different nature, since it requires that a MICTREE cell be able to realize organisms of any dimension (the largest possible organism in the implementation described herein consists of 16x16 cells). This challenge, which lies outside of the scope of this contribution, can be met by decomposing a cell into molecules and tailoring the structure of cells to the requirements of a given application [6,8].

Of course, current technology does not allow all these properties (universal construction, universal computation, self-repair) to be implemented, as in nature, by *physically* modifying the underlying hardware: they are implemented at a *logical* level by exploiting redundant spare hardware. The predicted development of technologies based on the manipulation of physical matter at the molecular level, however, will allow the realization of circuits of such complexity as to require the development of novel computational paradigms and architectures. The validation of the computational universality of these architectures is a fundamental step in their development: Turing machines are thus once again becoming a useful research tool in the field of computing.

References

1. M. Abramovici and C. Stroud. No-overhead BIST for FPGAs. In *Proceedings of First IEEE International On-Line Testing Workshop*, pages 90–92, 1995.
2. D. C. Ince, editor. *Mechanical Intelligence: Collected Works of A. M. Turing*, chapter Intelligent Machinery, pages 107–128. North-Holland, 1992.
3. S. C. Kleene. Turing's Analysis of Computability, and Major Applications of It. In R. Herken, editor, *The Universal Turing Machine a Half Century Survey*, pages 15–49. Springer-Verlag, second edition, 1995.
4. D. Mange. *Microprogrammed Systems: An Introduction to Firmware Theory*. Chapman & Hall, London, 1992. (First published in French as "Systèmes microprogrammés: une introduction au magiciel", Presses Polytechniques et Universitaires Romandes, 1990).
5. D. Mange, M. Goeke, D. Madon, A. Stauffer, G. Tempesti, and S. Duran. Embryonics: A New Family of Coarse-grained Field-Programmable Gate Array with Self-Repair and Self-Reproducing Properties. In E. Sanchez and M. Tomassini, editors, *Towards Evolvable Hardware*, pages 197–220. Springer-Verlag, Berlin, 1996.
6. D. Mange, M. Sipper, A. Stauffer, and G. Tempesti. Towards Robust Integrated Circuits: The Embryonics Approach. *Proceedings of the IEEE*, 88(04):516–541, April 2000.
7. D. Mange, A. Stauffer, and G. Tempesti. Embryonics: A Macroscopic View of the Cellular Architecture. In M. Sipper, D. Mange, and A. Prez-Uribe, editors, *Evolvable Systems: From Biology to Hardware*, volume 1478 of *Lecture Notes in Computer Science*, pages 174–184. Springer-Verlag, Berlin, 1998.

8. D. Mange, A. Stauffer, and G. Tempesti. Embryonics: A Microscopic View of the Cellular Architecture. In M. Sipper, D. Mange, and A. Prez-Uribe, editors, *Evolvable Systems: From Biology to Hardware*, volume 1478 of *Lecture Notes in Computer Science*, pages 185–195. Springer-Verlag, Berlin, 1998.

9. P. Marchal, P. Nussbaum, C. Piguet, S. Duran, D. Mange, E. Sanchez, A. Stauffer, and G. Tempesti. Embryonics: The Birth of Synthetic Life. In E. Sanchez and M. Tomassini, editors, *Towards Evolvable Hardware*, pages 166–196. Springer-Verlag, Berlin, 1996.

10. P. Marchal, A. Tisserand, P. Nussbaum, B. Girau, and H. F. Restrepo. Array processing: A massively parallel one-chip architecture. In *Proceedings of the Seventh International Conference on Microelectronics for Neural, Fuzzy, and Bio-Inspired Systems*, pages 187–193, Granada, Spain, April 1999.

11. E. J. McCluskey. *Logic Design Principles with Emphasis on Testable Semicustom Circuits.* Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

12. M. L. Minsky. *Computation: Finite and Infinite Machines.* Prentice-Hall, Englewood Cliffs, New Jersey, 1967.

13. H. F. Restrepo. *Implementation of a Self-Repairing Universal Turing Machine.* Swiss Federal Institute of Technology at Lausanne (EPFL), Lausanne, Switzerland, 2001. Ph.D. Thesis No 2457.

14. H. F. Restrepo and D. Mange. An Embryonic Implementation of a Self-Replicating Universal Turing Machine. In *Evolvable Systems: From Biology to Hardware (ICES01)*, volume 2210 of *Lecture Notes in Computer Science*, pages 74–87. Springer-Verlag, Berlin, October 2001.

15. H. F. Restrepo, D. Mange, and M. Sipper. A Self-Replicating Universal Turing Machine: From von Neumann's Dream to New Embryonic Circuits. In M. A. Bedau, J. S. McCaskill, N. H. Packard, and S. Rasmussen, editors, *Seventh International Conference on Artifivial Life*, pages 3–12. MIT Press, Cambridgre, August 2000.

16. M. Sipper, E. Sanchez, D. Mange, M. Tomassini, A. Perez-Uribe, and A. Stauffer. An Introduction to Bio-Inspired Machines. In D. Mange and M. Tomassini, editors, *Bio-Inspired Computing Machines: Toward Novel Computational Architectures*, pages 1–12. Presses Polytechniques et Universitaires Romandes, Lausanne, Switzerland, 1998.

17. A. Stauffer, D. Mange, M. Goeke, D. Madon, G. Tempesti, S. Durand, P. Marchal, and C. Piguet. MICROTREE: Towards a Binary Decision Machine-Based FPGA with Biological-like Properties. In *Proceedings of the International Workshop on Logic and Architecture Synthesis*, pages 103–112, Grenoble, France, December 1996.

18. G. Tempesti. *A Self-Repairing Multiplexer-Based FPGA Inspired by Biological Processes.* PhD thesis, Swiss Federal Institute of Technology, Lausanne, Switzerland, 1998.

19. G. Tempesti, D. Mange, and A. Stauffer. A robust multiplexer-based fpga inspired by biological systems. *Journal of Systems Architecture*, 43(10):719–733, 1997.

20. B. A. Trakhtenbrot. Comparing the Church and Turing Approaches: Two Prophetical Messages. In R. Herken, editor, *The Universal Turing Machine a Half Century Survey*, pages 557–582. Springer-Verlag, second edition, 1995.

21. A. M. Turing. On Computable Numbers, with an Application to the Entscheidungsproblem. *Proceedings of the London Math. Soc.*, 42:230–265, 1936.

22. J. von Neumann. *The Theory of Self-Reproducing Automata*. University of Illinois Press, Urbana, Illinois, 1966. Edited and completed by A. W. Burks.
23. H. Wang. A Variant to Turing's Theory of Computing Machines. *Journal of the ACM*, IV:63–92, 1957.
24. N. Wirth. *Programming in MODULA-2*. Springer-Verlag, Berlin, 1983.