ROMANIAN JOURNAL OF INFORMATION SCIENCE AND TECHNOLOGY
Volume **10**, Number 2, 2007, 145–156

Chomsky's Hierarchy & A Loop-Based Taxonomy for Digital Systems

G. ŞTEFAN

Politehnica University of Bucharest, Romania
& BrightScale, Inc., Sunnyvale, CA
E-mail: `gstefan@brightscale.com`

Abstract. Formal languages are supported by digital systems having a correspondent hierarchy. We show that Chomsky's taxonomy is paralleled by a *loop-based hierarchy* for digital systems. The machines used to recognize or generate a type n formal language belong to a specific class, differentiated from the one used for a type $(n-1)$ language (for $n = 1, 2, 3$). The difference between classes is given by an additional hardware loop closed in the last. Each new loop increases the autonomy of the system, making it able to support a more "expressive" language with a less restrictive grammar. We prove the correspondence between *type 3* languages and *2-loop* machines, between *type 2* languages and *3-loop* machines, and between *type 1* languages and *4-loop* machines.

1. Introduction

The taxonomy proposed by Noam Chomsky [2, 3, 4] fits perfect with the actual formal languages, without providing any idea about a **unique** *guiding principle* working behind this hierarchy. Maybe there is no such a unique principle. Or maybe there are some indirect clues to be emphasized starting from the way the associated machines are structured. Consequently, we intend to provide an indirect solution, as a first step on the way to find the supposed guiding principle. The guiding principle working for the associated machines will provide an *indirect* answer to our legitimate question: "what is that *unique* criterion able to explain the difference between a type n language and a type $(n-1)$ language (for $n = 1, 2, 3$)?"

The well known correspondence between the language type (*regular, context-free, context-sensitive*) and the associated machine (*finite automata, push-down automata, linear-bounded automata*) helps us to move the search of the guiding principle from the domain of languages into the co-domain of the machines.

The next section is about how a digital system grows in size or gains new features. A taxonomy of digital systems, based on the featuring mechanism, is presented. The third section shows the direct correspondence between the language type and the minimum number of the loops included each other inside the associated machines. The fourth part contains comments on the number of loops, on machine complexity, and on language efficiency. A concluding section ends this paper.

2. Growing by Compositions & Featuring by Loops

A digital system computes partial recursive functions [6]: $f : \{0,1\}^n \to \{0,1\}^m$. It uses basic functions (*initialization, increment, selection*) and applies rules (***composition, primitive recursiveness, partial recursiveness***). For the basic functions, linear-size and poly-log-time combinational solution are provided. There are two mechanisms governing the structuring process in digital systems: growing the *size*[1] and adding *features*. The first is driven by the ***composition rule*** and the second by both, ***primitive & partial recursiveness.***

In Figure 1 we present the physical structure associated with the composition defined as:

$$f(x_0, \ldots x_{n-1}) = g(h_0(x_0, \ldots x_{n-1}), \ldots h_{m-1}(x_0, \ldots x_{n-1})).$$

A digital system grows by repeated application of this rule for different n and m. The functionality of the resulting system remains in the same class as the functionality of the composed subsystems.

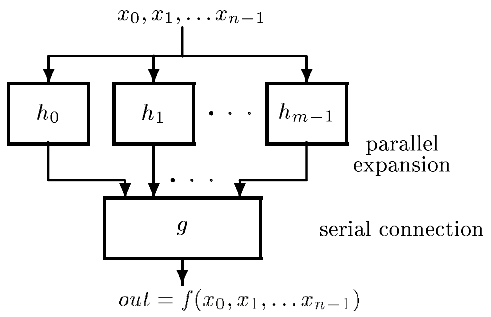
$x_0, x_1, \ldots x_{n-1}$
h_0
h_1
h_{m-1}
parallel expansion
g
serial connection
$out = f(x_0, x_1, \ldots x_{n-1})$

Fig. 1. Implementing the composition rule. The composition of the function g with the functions $h_0, \ldots, h_{m-1}$ means a two level system. Results a two-dimension expansion rule in digital systems. The functions $h_0, \ldots, h_{m-1}$ are *parallel* expanded, and the function g represents a *serial* expansion.

[1]We consider the *size* of a machine (circuit) as distinct from its *complexity*. Size is expressed in the number of elementary components (2-input gates, for example) used to build the system. The complexity refers to the minimal size of the description (number of symbols, for example) used to specify a system. This definition of complexity is inspired by the Chaitin's *algorithmic complexity* [1]. A *complex system* has its size in the same magnitude order with its complexity ($S_{complex\_system} \sim C_{complex\_system}$). See details in [9], [10].

New kinds of functionality occur only when recursiveness is involved. Primitive recursiveness introduces ***data loop*** and partial recursiveness asks for the ***control loop***. Data loop optimize the size of the circuit, while the control loop is mandatory for "directing" the computational process.

The operation of closing loops adds new features in a digital system. The no-loop, combinational family of circuits are able to solve all aspects of computation if no actual solutions are considered. But for real and optimal solutions successive loops are closed generating a circuit hierarchy. Thus, the loop-based taxonomy of digital systems (see for details [9]) generates the following open list of systems:

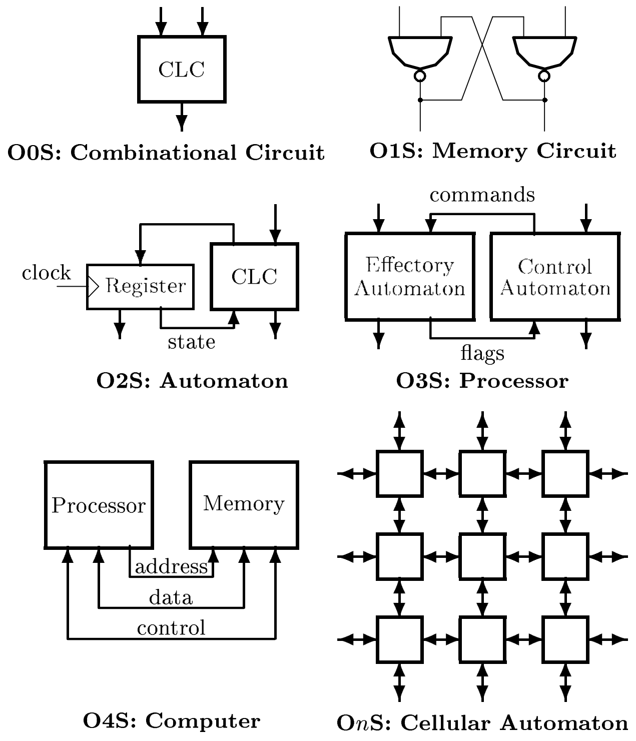

Fig. 2. Examples of circuits belonging to different orders. A combinational, no-loop circuit is an O0S. A memory, one-loop circuit is O1S. Because the register belongs to O1S, closing a loop containing a register and a combinational circuit (which is an O0S) results an automaton: a circuit in O2S. Two loop-connected automata – a circuit in O3S – works as a processor. An example of O4S is a simple computer obtained by loop connecting a processor with a memory. The cellular automaton contains a number of loops related with the number of automata it contains.

0-loop combinational systems we call *order 0 systems* (O0S), include all kinds of no-loop gate networks (adders, multiplexors, decoders, ...)

1-loop memory systems we call *order 1 systems* (O1S), include all circuits with one level of loops closed inside them (latches, clocked latches, random access memories, file registers, master-slave flip-flops, registers, ...)

2-loop automata systems we call *order 2 systems* (O2S), include systems with two level of loops closed inside, one for providing the memory function and the second allowing autonomous behaviors (JK flip-flops, T flip-flops, counters, automata, finite automata, file registers with ALU, ...)

3-loop processing systems we call *order 3 systems* (O3S), include systems with the third loop closed over systems already containing 2 loops (counter automata, stack automata, simple RISC processors, ...)

4-loop computing systems we call *order 4 systems* (O4S), include systems with the fourth loop closed over an O3S (microprogrammed processors, microcontrollers, simple computers, ...)

...

***n*-loop self-organizing systems** we call *order n systems* (O*n*S), include systems with the number of internal loops in the same order of magnitude as the size of the system (cellular automata, complex neural networks, ...).

Although families of O0Ss have the competence to solve any computation, loops are used to provide new features very helpful in providing effective solutions. Each new loop adds a certain degree of autonomy which improves the "ability" of the system to solve efficiently more complex problems. In this paper will be proved that each new loop is associated with the removal of a restriction applied to the set of productions defining a generative grammar.

3. Grammar Type & Number of Loops

In this main section we aim to prove the consistency of the featuring mechanism in digital systems (shortly presented in the section 2) with another hierarchy emphasized in a related domain: Chomsky's formal language theory. Some important things happen when a new loop is added in a digital system, because it is the way to move from a machine associated with a type of formal language toward the machine associated with the next more "expressive" language in the hierarchy. Let us examine this strange effect of the correlation between the machine's *autonomy* and the *expressiveness* of the language.

A short review on Chomsky's hierarchy of grammars is helpful. A grammar is defined by the following finite sets of symbols: non-terminals $N = \{S, A, B, \ldots\}$, where S is the special starting symbol, terminals $T = \{a, b, \ldots\}$, and productions, P, having the form $L \rightarrow R$, where $L, R \in \{N \cup T\}^*$.

Definition 1. According to Chomsky's taxonomy there are the following type of generative grammars introduced by adding in each step a new restriction to the way the productions are defined. Results:

type 0 grammars generate *recursively enumerable languages*, $\mathcal{L}_0$, using rules with the restriction: $L =$ `any string with at least one non-terminal`

type 1 grammars generate *context-sensitive languages*, $\mathcal{L}_1$, using rules with the additional restriction: $R =$ `any string no shorter than` L (in the generation process the string cannot be shortened)

type 2 grammars generate *context-free languages*, $\mathcal{L}_2$, using rules with the supplementary restriction: $L =$ `one symbol` (which must be, according to the first restriction, a non-terminal)

type 3 grammars generate *regular languages*, $\mathcal{L}_3$, using rules with the supplementary restriction: $R = x|xY$, where $x \in T$ and $Y \in \{N\}^*$ (the terminals are generated segregated from the non-terminals, or the stream of symbols grows at one end only).

Each new restriction applied to the way productions are defined reduces the "expressivity" of the associated language.

Also, there is a very well known correspondence between formal languages and the associated machines. All these machines are finite automata plus a sort of memory. The more featured is the memory, the less restrictive is the associated language.

- $\mathcal{L}_3 \leftrightarrow$ *Finite Automata* (FA): a machine with cycle level memory (it stores only what can be stored in one finite word called *state*)
- $\mathcal{L}_2 \leftrightarrow$ *Stack Automata* (SA): a finite automaton with stack memory, a destructive-read memory (the stack memory stores a stream of data, but it "forgets" the information when it delivers it)
- $\mathcal{L}_1 \leftrightarrow$ *Linear Bounded Memory Automata* (LBA): a finite automaton with finite non-destructive read memory (or an SA with an additional stack to save the data read from the main stack)
- $\mathcal{L}_0 \leftrightarrow$ *Turing Machine* (TM): a finite automaton with infinite non-destructive read memory.

Each actual machine is defined, and has the complexity given by a finite Boolean table defining the *transition function* of the associated finite automaton. The resulting function is implemented as a combinational circuit. The rest consists in different simple *memory functions*: registers, stacks, queues (in Post's definition), list memory (in Turing's definition), random access memory (in other equivalent definitions), ... Let's see how from FA to LBA we can go only closing appropriate loops over simple subsystems.

3.1. Type-3 Grammars & 2-Loop Machines

The goal of this subsection is to prove that an optimal digital system must have *at least two* internal loops for recognizing or generating the regular (type 3) languages.

In textbooks on formal languages it is stated that *each type 3 language can be recognized/generated by the final states of an initial deterministic automaton* (see, for example [5]). Indeed, according to the way the productions are defined, they can be reduced to the following two forms: $A \rightarrow aB$ and $A \rightarrow a$. At each generating step, one terminal (a) is added to the string, and it is preserved (for $A \rightarrow aB$) or not (for $A \rightarrow a$) the possibility of a new similar action. Therefore, each symbol in a well formed string depends *only* by its predecessor. The value of the last received/generated symbol can be "memorized" by the internal state of a finite automaton. Each state of the automaton can say something only about the last processed symbol and that is enough. Because the sets N, T, and P are finite, the automaton that recognizes any language $L(G) \in \mathcal{L}_3$ can be defined as a finite machine.

Theorem 1. *The lowest order of a system that implements any finite automaton is two.*

Proof. The kernel of a finite automaton is the associated half-automaton defined by the following triplet:

$$HA = (X, Q, f)$$

where: X is the finite input set, Q is the finite set of the states, and $f : X \times Q \rightarrow Q$ is the state transition function. The general structure of a half-automaton is presented in Figure 3, where:

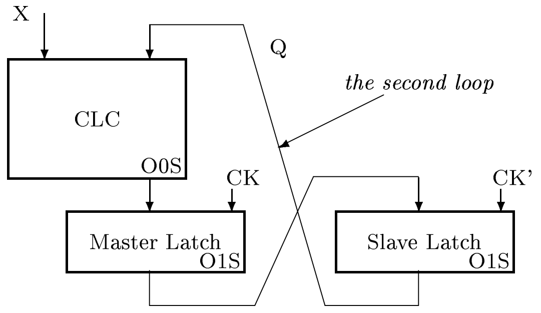

Fig. 3. The internal structure of a half-automaton. The second loop is closed through two *simple* latches (containing the first level of loops) and a *complex* combinational logic circuit.

- *CLC* is a combinational logic circuit that computes the transition function f;
- *Slave Latch* is a collection of one-bit latches that *store* the current state;
- *Master Latch* is another latch that allows to close properly the loop over the entire system (it has only an electrical function allowing the synchronous behavior of the system).

In this system there are two level of loops:

- the *first loop* level in each one-bit latch, allows the *storing* function;

– the *second loop* level is imposed by the *state transition function f* which is defined in $X \times \mathbf{Q}$ and with values in $\mathbf{Q}$ (the current value of the variable Q determines the next value of **the same** variable). □

In the context of regular language processing we can say that the two levels of loops are necessary and sufficient to manage regular languages because: (1) the first loop is used to build the circuit that *stores* the last received or generated symbol: the slave latch from the state register; (2) the second loop, closed through register and the combinational circuit, is for *sequencing* the process of recognition/generation. No more memory is needed because the productions are very simple. The finite automata are the simplest and the smallest digital machines that recognize and generate regular strings. We can define a more structured optimal machine, but never a less structured optimal one having the order 1 or 0.

3.2. Type-2 Grammars & 3-Loop Machines

We are expecting that the step towards the more expressive languages belonging to $\mathcal{L}_2$ should require a more autonomous machine to recognize or to generate them. Can automata work at the level of context-free languages? Yes, they work, but not as *finite* automata, because only the "infinite" automata can do the job. If we don't agree with "infinite" automata, then a third order system must be used. If we wish to use an "infinite" automaton to recognize strings belonging to the second type language, then an automaton having $|Q| \in O(n)$ must be used, where: n is the length of the string and $|Q|$ is the dimension of Q. But our aim is to keep the complexity at the lowest possible level. In this respect we must find a circuit solution having its complexity constant, i.e., independent of n.

Let us start with a short discussion about the classical example offered by the language $\{a^n b^n | n > 0\} \in \mathcal{L}_2$. If we want to recognize this language using an automaton, then the problem raised is to know how many of as are received before the first occurrence of b. The machine must memorize somewhere the number of as. This place for an automaton is its "state space", but in this case the automaton becomes an "infinite" machine. The solution is an additional memory.

Going back to textbooks we find that *two type languages can be recognized/generated by pushdown automata.*

A simple remark: PDA is an efficient machine because the automaton is a complex finite machine and the stack is an infinite but simple, recursively defined machine.

Theorem 2. *The lowest order of a system that implements a push-down automaton is three.*

Proof. Because the push-down automata is build using a finite automaton loop coupled with a push-down stack, it is a third order system. Indeed, a finite automaton is a second order system and the push-down stack has the same order because it is a simple, recursively defined "infinite" automaton. It is an automaton because its simplest implementation (see Figure 4) needs a register loop connected with a multiplexor. If the content of the stack (stored in Register (see Figure 4)) is $\{s_0, s_1, s_2, \ldots\}$, then: *data_out* $= s_0$, and for $\{push, pop\} = \{1, 0\}$, and *data_in* $= x$ the multiplexor selects to the input of Register $\{x, s_0, s_1, \ldots\}$, for $\{push, pop\} = \{0, 1\}$ selects to the input of Register $\{s_1, s_2, s_3 \ldots\}$, and for $\{push, pop\} = \{0, 0\}$ the multiplexor selects to the input of Register $\{s_0, s_1, s_2 \ldots\}$. □

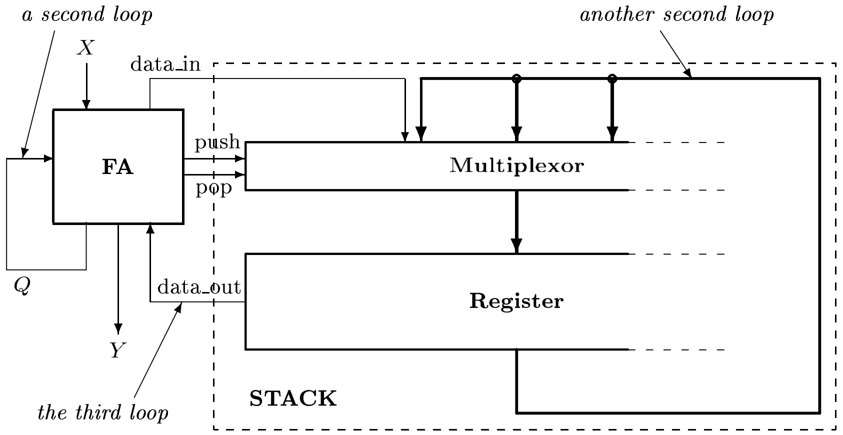

Fig. 4. Push-Down Automata. The stack belongs to O2S because it is a simple, recursive defined, "infinite" automaton (a register (O1S) loop connected with a multiplexor (O0S)). Results PDA is an O3S.

The stack is the simplest memory device because: (1) it stores only one string; (2) the string is accessed only to one of its ends (last-in-first-out); (3) the read operation is destructive (the memory forgets the information read because of the access type). The simplicity is the reason for using this memory to build the first machine more complex than a finite automaton. But, the same simplicity is the reason for which we must renounce to this memory if we want to approach the next type of languages. For the context-sensitive languages we need a memory in which we can access many times the same stored content.

3.3. Type-1 Grammars & 4-Loop Machines

When we try to build a machine associated to the language $\{a^n b^n c^n | n > 0\} \in \mathcal{L}_1$ we must add a supplementary memory resource in order to deal with the *c*s. This leads us toward the *third loop.*

In the general case, we can use for languages belonging to $\mathcal{L}_1$ a *finite defined machine* only by adding, to the pushdown stack automaton, a new push-down stack [7] (see Figure 5). In this case a new loop is closed in the machine and it becomes a *fourth order system.* The new stack is used to compensate the limitation of the stack memory which forgets when it is read. Thus, the simplest automaton having associated a non-destructive memory is made by adding a new pushdown stack to a PDA. For each $POP0$ from STACK 0 a $PUSH1$ with the read symbol in STACK 1 is performed. For each $POP1$ from STACK 1, a corresponding $PUSH0$ can be made in STACK 0. The sizes of each stack can be linearly bounded to the string length.

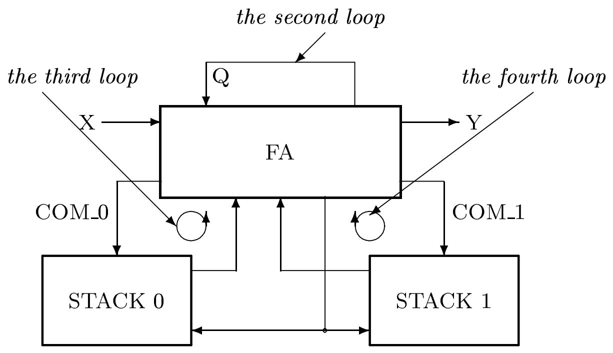

Fig. 5. PDA with stack memory. *FA & STACK 0 = PDA.* Adding STACK 1 a fourth loop is closed. Any symbol read from STACK 0 can be inserted in STACK 1 and viceversa. *STACK 0 & STACK 1* behaves like a list memory.

The *fourth loop* of the system is necessary because it gives us access to a new external memory. This additional memory was imposed because another restriction has been removed when $\mathcal{L}_1$ enters the scene.

It is known that *context-sensitive languages are recognized/generated by linear bounded automata.*

Theorem 3. *The lowest order of a system that implements a linear bounded memory automaton is four.*

Proof. The simplest memory having non-destructive reading can be made by loop connecting two push-down stack memories. Because a push-down stack is a second order system, a memory with non-destructive reading is a third order system and the resulting system is a fourth order system (see Figure 5). An equivalent structure is presented in Figure 6, where:

- FA is a finite automaton (a second order system)
- $U/D\ COUNTER$ (another second order system) is an "infinite" automaton with a simple structure (its complexity is $C_{U/D\ COUNTER} \in O(1)$, even if its size is $S_{U/D\ COUNTER} \in O(logn)$) used to point a symbol in memory
- RAM is a random access memory (a first order system) used to store the string (its complexity is $C_{RAM} \in O(1)$, even if its size is $S_{RAM} \in O(n)$)

The resulting structure has also two loops closed over a finite automaton. □

The hardware requirement for context-sensitive languages implies a more structured and a more functional segregated machine. This machine has two supplementary loops added to an automaton with two distinct roles: (1) the first, through *RAM*, in order to access an external *memory support*; (2) the second, through *U/D COUNTER* and *RAM*, in order to access an external *memory function*: a *bi-directionally scanned list.*

The *list* can support a more complex computation than the *stack*. Both are strings, but the second allows only a limited and destructive access to the content of the string. In a memory hierarchy the list has a higher "order" because it is implemented by two loop connected stacks.

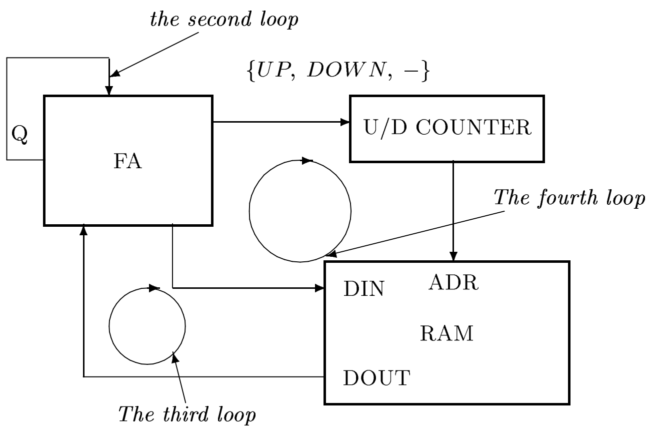

Fig. 6. Automaton with Linear Bounded Memory. FA uses the loop connected RAM as storing device. The fourth loop closed through the counter organizes the RAM's content as a list.

4. Loops & Complexity

An automaton dealing with an $\mathcal{L}_2$ language has the size of its space state in $O(n)$, where n is the maximum length of the string to be processed. Accordingly, the complexity of the associated combinational circuit becomes unmanageable. The problem is solved introducing a new concept and a new machine: PDA. What we did substituting an "infinite" automaton with a PDA? We segregated the "hidden" simple part of the "infinite" automaton from its inherent complex part. The stack is a big, but simple subsystem, and the remaining finite automaton is a small, but complex subsystem. Because they are loop connected the resulting machine exert an increased autonomy without a significantly increased complexity.

Thus, adding a new loop, the *big* & *simple* part – the stack – and the *small* & *complex* part – the finite automaton – are very clearly emphasized. Results an optimal structure, where the complexity is minimized and the big parts are simple.

It looks like the closing of a new loop helps to increase the complexity of processing maintaining the complexity of the machine at a similar, low level. Indeed, the only complex structure in PDA or LBA is the finite automaton, more precisely its combinational circuit. All the other components involved in defining different machines (registers, multiplexors, counters, RAMs) are **simple** because they have constant sized, recursive definitions. The size of their definitions does not depend on the dimensions of the sets N, T, and P, or on the maximum length of the recognized or generated strings of symbols. Fortunately, the definition's size for the finite automaton depends only on the finite sizes of the sets N, T, and P.

The list of meaningful machines ends with the Turing Machine (TM). It can be represented as the LBA with a very important adagio: both, the dimension of the counter and of the memory are considered infinite. The non finiteness of the memory is needed because the restriction "$R =$ `any string no shorter than` L" does not act, thus, the intermediary length of a string during its generation can not be predicted, and consequently can be bigger than the final form.

The path from the families of combinational circuits toward the TM has its important milestones marked each time by a new level of loops. The *autonomy* of the resulting systems increases with each additional loop. This process culminates in the possibility of building a *0-State Universal TM*: the simplest possible computational structure (see [10]). In a 0-State Universal TM the segregation between the complex part and simple part is maximal. Indeed, the complex part of programs stored in memory are completely segregated by the simple part of the hardware designed using only simple, recursively defined structures.

The first consequence of closing loops is the possibility to segregate simplicity from complexity, thus reducing the actual complexity of the machines involved in computation.

The second consequence, related with the first, is the increased autonomous behavior of the system when it gains a new loop. The autonomy of the machines evolves in parallel with the "expressivity" of the associated languages.

5. Concluding Remarks

1. The actual structure of machines associated with formal languages tends to minimal complexity. In this respect we can state the following proposition: *Digital machines that recognize and generate formal languages can be "infinite" (big sized) machines, but they must have finite definitions (small complexity).*

2. The most important conclusion of this paper is that there exists a direct correspondence between:

- $\mathcal{L}_3 \leftrightarrow$ **O2S**
- $\mathcal{L}_2 \leftrightarrow$ **O3S**
- $\mathcal{L}_1 \leftrightarrow$ **O4S**

The unique principle governing the loop-based taxonomy can be mirrored by a similar unique principle in language hierarchy? This remains for the time being an open question.

TM does not have associated a distinct order in the presented structural hierarchy of digital systems, because it is only a pure theoretical model.

Between the context-sensitive, type 1 languages and the type 0 languages there are many other types of languages, corresponding to less restricted productions defining their grammars. These languages are out of our interest because the majority of programming languages are context-free. Systems having the order more than four are associated to these hypothetical languages.

3. The 0-state, control free Universal TM shows us that the evolution of the computing machines converted the *hardware* complexity into the *software* complexity. Nowadays VLSI technologies can build big sized circuits only if they are simple. *The structural complexity cannot grow with the same speed as the size.* We must avoid growing the complexity in order to preserve our ability to build very large circuits. Fortunately, the loops closed inside the digital systems provide a lot of help in this respect.

References

[1] CHAITIN G., *Algorithmic Information Theory*, IBM J. Res. Develop., July, 1977.

[2] CHOMSKY N., *Three Models for the Description of Languages*, IEEE Trans. on Information Theory, **2**:3 , 1956.

[3] CHOMSKY N., *On Certain Formal Properties of Grammars*, Information and Control, **2**:2, 1959.

[4] CHOMSKY N., *Formal Properties of Grammars*, *Handbook of Mathematical Psychology*, Wiley, New York, 1963.

[5] HOPCROFT J., MOTWANI R., ULLMAN J., *Introduction to Automata Theory, Languages, and Computation*, 2^{nd} Edition, Addison Wesley, 2001.

[6] KLEENE S. C., *General Recursive Functions of Natural Numbers*, Math. Ann., **112**, 1936.

[7] MINSKY M., *Recursive Unsolvability of Post's Problem of 'Tag' and Other Topics of Turing Machines*, Annals of Mathematics, **74**, pp. 437–455, 1961.

[8] SHANNON C. E., *A Universal Turing Machine with Two Internal States*, Annals of Mathematics Studies, **34**: Automata Studies, Princeton Univ. Press, pp. 157–165, 1956.

[9] ŞTEFAN G., *Loops & Complexity in Digital Systems. Lecture notes on digital design in Giga-Gate per Chip Era*, posted at `http://arh.pub.ro/gstefan/`.

[10] ŞTEFAN G., *A Universal Turing Machine with Zero Internal States*, Romanian Journal of Information Science and Technology, vol. **9**, no. 3, 2006, pp. 227–243.

[11] TURING A. M., *On Computable Numbers with An Application to the Eintscheidungsproblem*, Proc. London Mathematical Society, **42** (1936), **43** (1937).