information

MDPI

Article

Triadic Automata and Machines as Information Transformers

Mark Burgin

Department of Mathematics, University of California, Los Angeles, 520 Portola Plaza, Los Angeles, CA 90095, USA; mburgin@math.ucla.edu

Received: 12 December 2019; Accepted: 31 January 2020; Published: 13 February 2020

Abstract: Algorithms and abstract automata (abstract machines) are used to describe, model, explore and improve computers, cell phones, computer networks, such as the Internet, and processes in them. Traditional models of information processing systems—abstract automata—are aimed at performing transformations of data. These transformations are performed by their hardware (abstract devices) and controlled by their software (programs)—both of which stay unchanged during the whole computational process. However, in physical computers, their software is also changing by special tools such as interpreters, compilers, optimizers and translators. In addition, people change the hardware of their computers by extending the external memory. Moreover, the hardware of computer networks is incessantly altering—new computers and other devices are added while other computers and other devices are disconnected. To better represent these peculiarities of computers and computer networks, we introduce and study a more complete model of computations, which is called a triadic automaton or machine. In contrast to traditional models of computations, triadic automata (machine) perform computational processes transforming not only data but also hardware and programs, which control data transformation. In addition, we further develop taxonomy of classes of automata and machines as well as of individual automata and machines according to information they produce.

Keywords: information; automaton; machine; hardware; software; modification; process; inductive; recursive; superrecursive; equivalence

1. Introduction

It is well known that computers are processing information. At the same time, they are also containers or carriers of information. That is why it is so important to study properties of computers and computer networks from the information perspective. To efficiently do this, researchers need adequate models of computers and computer networks. Many properties of computers, computer networks and computation are well presented in conventional models of automata and algorithms. However, there are still properties that demand new models.

Traditionally, computation is treated as data transformation, which modifies information contained in data and creates new information. At the same time, there were important traits of computers and computer networks, which were missed in traditional models. For instance, the majority of these models do not have input and output systems. The absence of these systems prevented finding and formalizing inductive computations for a long time [1,2].

Other ignored characteristics are related to hardware and software transformations, which take place in computers and computer networks. As a matter of fact, in physical computers, their programs are changing by special software tools such as interpreters, compilers, optimizers and translators. Besides, by using external memory, people change the hardware of their computers. The

hardware of computer networks is permanently altering—new computers and other devices are added while other computers and other devices are disconnected.

The goal of this paper is to build models of computation reflecting these characteristics of physical information processing systems and to study their properties.

It is necessary to remark that in our study, we make a distinction between automata, which works autonomously, and machines, which can involve participation of people.

Looking at the history of theoretical computer science, we can see that in the simplest form, hardware modification was present in models of computation from the very beginning. Indeed, to preserve finiteness of its memory, a Turing machine had to modify its hardware or more exactly, its memory in the process of computation because in a general case, the length of the used tape could become larger than any number as the length of the computed words increased. To achieve this, a Turing machine had to add new cells to its tape, i.e., the memory was changing.

This property of Turing machines essentially increases their power because with a finite tape (memory) such a machine would be equivalent to, i.e., not more computationally powerful than, a finite automaton.

In such a natural way, hardware modification emerged at the very beginning of the theory of computation. However, the first model, in which hardware was essentially changing in the process of computation, was *hardware modification machine* [3–5]. Such a machine is a collection of units—each of which is a multiple-input finite-state transducer—and a system of arbitrary connections between units. These connections can be changed during computation, and new units can be activated as the computation proceeds.

Even more essential software modification is performed in inductive Turing machines, where its memory is fundamentally constructed by a special agent in the form of a Turing machine or another inductive Turing machine [6]. Treating this agent as a component of the main machine, we obtain an abstract automaton with hardware self-modification because memory is a part of the machine hardware.

While physical computers also perform operations with their programs using special software tools such as interpreters, compilers, optimizers and translators, the concept of software modification in abstract automata came to computer science much later. The idea of such modification for increasing power of computations belongs to Stephen Kleene, who formulated a conjecture that it might be possible that algorithms that change their programs while computing would be more powerful than Turing machines [7].

To test this conjecture, Mark Burgin constructed reflexive Turing machines [8]. This was the first theoretical model of algorithms that change their programs while computing. Using this model, Burgin proved that the class of reflexive Turing machines is computationally equivalent to the class of Turing machines, i.e., both classes of abstract machines have the same computing power [8]. In such a way, Kleene's conjecture was disproved but, at the same time, it was proved that reflexive Turing machines can be essentially more efficient than Turing machines. Namely, a relevant reflexive Turing machine can effectively outperform any Turing machine that computes the same function [8,9].

A more general model of automata modifying their software—symmetric Turing machines or S-machines—was later suggested by Marcin Schroeder [10,11]. The concept of a symmetric inductive Turing machine was introduced in [12].

There are also directions in computer programming, such as reflective programming or metaprogramming, which allow observing and modifying computer programs at runtime [13–15].

Here, we synthesize hardware and software modification in one theoretical model of a triadic automaton or triadic machine, which processes and changes not only data (infware) but also the software and the hardware of itself. Thus, in general, a triadic automaton (triadic machine) is able to transform all three of its basic components.

Thus, triadic automata (triadic machines) transform information of three types because the hardware of the automaton (machine) is physical information container (carrier), its software is textual information container (carrier) and infware is symbolic information container (carrier).

This paper has the following structure. In the second section, after Introduction, we consider relations between algorithms, automata and machines. In the third section, we discuss the inner structure of abstract automata, which consists of its hardware, software and infware, which can be treated as data processed by this automaton. In the fourth section, we introduce and formally describe the concept of a triadic automaton. In the fifth section, we study the dynamics of triadic automata and machines. In Conclusion, we summarize obtained results and suggest directions for the future research.

2. Algorithms, Automata and Machines

It would be no exaggeration to say that the basic concepts of computer science are algorithms, automata, machines and computation. Nevertheless, there are no unanimously accepted definitions of these important concepts. That is why, in this section, we analyze these concepts, explicate relations between them and suggest informal definitions of these concepts in the context of information processing.

Definition 2.1. An *algorithm* is a constructive finite description of a set of processes aimed at solving some problem with the exact definitions of its input and result.

Here, constructive means that all described operations are comprehensible, executable and finite.

There are different types, categories, kinds, forms and classes of algorithms. Let us consider some of them.

According to the world structuration described in [16], the following types of algorithms are used by people:

1. *Physically represented algorithms*, e.g., the hardware of computers;
2. *Structurally represented algorithms*, e.g., structures of computer programs or of transition rules of finite automata;
3. *Mentally represented algorithms*, e.g., mental schemas of addition or multiplication [17–19].

In turn, physically represented algorithms have the following forms:

1. *Instrumental algorithms*, e.g., algorithms in the form of an automaton;
2. *Textual algorithms*, e.g., a system of instructions or rules;
3. *Numeric algorithms*, e.g., algorithms as weights in neural networks.

Textual and numeric algorithms together form the class of *symbolic algorithms*.

In this context, textual algorithms can be divided into three subclasses:

1. *Operationally expressed* textual algorithms, e.g., systems of instructions;
2. *Functionally expressed* textual algorithms, e.g., partially recursive functions;
3. *Intentionally expressed* textual algorithms giving only descriptions of what is necessary to do, e.g., programs in functional or relational programming languages.

This classification reflects levels of explicitness of algorithm representations.

According to their forms, the following categories of algorithms are used by people:

1. *Parametric algorithms*, e.g., weights in neural networks;
2. *Instruction algorithms*, e.g., rules in Turing machines;
3. *Description algorithms*, e.g., programs in functional or relational programming languages.

This classification reflects levels of symbolism in algorithm representations.

There are also levels of algorithms, which reflect grades of explicit descriptions of computational processes by algorithms [20,21]. For simplicity, we describe these levels for algorithms that have the form of a system of instructions or rules.

Algorithms of the first level contain only instructions (rules) for data transformation.

Algorithms of the second level contain instructions (rules) for data transformation and execution instructions (or metarules), which describe how to apply data transformation instructions (rules).

For instance, in finite automata, instructions (rules) that define how to select an appropriate transition are execution instructions (or metarules).

Note that metarules of instruction selection can essentially influence the functioning of systems where they are used.

Algorithms of the third level contain instructions (rules) for data transformation, execution instructions (metarules) of the first level, which describe how to apply data transformation instructions (rules) and execution instructions (metarules) of the second level, which describe how to apply execution instructions (metarules) of the first level.

It is possible to continue this construction considering algorithms of any level n.

Algorithms are utilized to control and direct the functioning of automata and machines, which are systemic devices. That is why, at first, we consider the general concept of a device.

Definition 2.2. A device is a structure that performs actions and generates processes.

There are different types, categories, kinds and classes of devices. Let us consider some of them.

According to the world structuration described in [16], the following types of devices are used by people:

1. *Physical devices*, e.g., computers or calculators;
2. *Abstract devices*, e.g., abstract automata such as finite automata or Turing machines;
3. *Mental devices*, e.g., mental schemas of neural systems, which perform addition or multiplication [22,23].

It is important to understand that only instrumental algorithms are devices. Two other classes—textual and numeric algorithms—are not devices because they do not perform actions themselves. For instance, a Turing machine is a device while a partial recursive function is not a device. Algorithms that are not devices need devices or people to be performed.

According to their origin, the following classes of devices are created and used by people:

1. *Artificial devices* are created by people, e.g., computers;
2. *Natural devices* exist in nature, e.g., an organism of a living being or the Earth, considered as a device;
3. *Combined devices* are combinations of artificial and natural devices, e.g., a car with a driver or a plane with a pilot.

People construct and use a diversity of artificial devices—computers, cars, planes, cell phones, ships and so on. At the same time, people use also different natural devices. The system of a sundial is an example of such a device. It consists of three subsystems: the Sun, the Earth and the sundial itself.

According to the classification of natural systems, the following kinds of devices exist:

1. *Animate devices* are or include living individuals,
2. *Inanimate devices* do not include living individuals, and
3. *Hybrid natural devices* are combinations of animate and inanimate devices.

In some generalized sense, it is possible to consider all living beings as animate devices. The Solar System is an example of inanimate device.

Automata and machines are important special cases of devices.

Definition 2.3. An *automaton* is a device that autonomously performs actions prescribed by an algorithm when a relevant input is given.

The functioning of automata is controlled by algorithms.

Similar to devices in general, there are three basic classes of automata:

1. *Artificial automata* are created by people, e.g., electronic clocks;
2. *Natural automata* exist in nature, e.g., the Solar system;

3. *Combined automata* are combinations of artificial and natural devices, e.g. a sundial, which consists of two natural systems - the Sun and the Earth – combined with one artificial system - the sundial itself.

.

In essence, all classifications of devices are applicable to automata.

Very often people treat theoretical automata and machines as the same class of objects. For instance, Turing machines are considered the most popular class of abstract automata. However, in our study, we make a distinction between automata and machines.

Definition 2.4. A *machine* is a device such that when a relevant input is given, performs actions prescribed by an algorithm which can involve participation of people.

For instance, cell phones are machines while clocks are automata.

Note that according to these definitions, any automaton is a machine, but it is not true that any machine is an automaton.

In essence, all classifications of devices induce corresponding classes of automata and machines because automata and machines are devices. To avoid repetition, we do not consider those types, categories, kinds and classes of automata and machine classification that are induced by classifications of devices. At the same time, there are classifications of automata and machines brought about by classification of algorithms. It is possible to take the following classification as an example.

According to the classification of algorithms that control automata, the following categories of automata are used in computer science:

1. *Parametric automata*, e.g., neural networks, are controlled by parametric algorithms;
2. *Instruction automata*, e.g., Turing machines, are controlled by instruction algorithms;
3. *Description automata*, e.g., symmetric Turing machines in the sense of [10,11,24], are controlled by description algorithms.

Devices in general and automata and machines, in particular, can perform various functions and generate diverse processes. Here we are interested in information processing in the form of computation.

Although computation pervades the contemporary society, there is no unanimously accepted definition of computation [25]. At the same time, there is a variety of different definitions and description—some of which we consider here.

After Turing machine was accepted as the uppermost model of algorithms, computation has been interpreted as what a Turing machine is doing. When it became clear that computation can go beyond Turing machines, the opposite trend appeared, in which it was supposed that any system in nature or society is computing. This approach is called pancomputationalism (cf., for example, [26 – 28]).

As it is explained in [29], there are three basic levels of generality in understanding the phenomenon of computation:

On the top level, *computation* is perceived as any transformation of information and/or information representation.

On the middle level, *computation* is distinguished as a discrete process of transformation of information and/or information representation.

On the bottom level, *computation* is recognized as a discrete process of symbolic transformation of information and/or symbolic information representation.

Here, we take on the engineering approach to computation according to which computation is a process performed by information processing devices. Besides, in what follows, we consider only computing automata and machines.

3. Inner Structure of Abstract Automata

On analyzing a computer, we see that it has various devices, e.g., one or several processors, memory of different types and input/output devices [30]. All these devices and connections between them constitute the *hardware* of this computer.

In addition, a computer has various programs, which direct its functioning. All these programs cast the *software* of this computer.

Besides, a computer works with diverse data. These data are unified under the name the *infware* of this computer.

In a similar way, the inner structure of an abstract automaton consists of three components—hardware, software and infware—and relations between them [1].

Definition 3.1. The *hardware* of an abstract automaton comprises theoretical (abstract) devices, such as a control device, processor or memory, which play a part in computations performed by this automaton, and connections between these devices.

In a general situation, the hardware of a real information processing system has three key components: the input subsystem, output subsystem and processing subsystem. Thus, to properly represent information processing systems, which are actually utilized by people, the hardware of an abstract automaton must have three basic components: (abstract) *input device*, (abstract) *information processor*, and (abstract) *output device* [1]. In many theoretical models of computation, input and output devices are either not specified or represented by components of the common memory and/or of the processor. For example, in a typical Turing machine, operations of input and output utilize the working memory—one or several tapes. In contrast to this, inductive Turing machines contain special input and output registers, e.g., tapes [6]. The same is true for pointer machines. Indeed, a pointer machine receives input—finite sequences of symbols (words) from its "read-only tape" (or an equivalent storage device) -- and it writes output sequences of symbols on an output "write-only" tape (or an equivalent storage device).

Neural networks also have these core components: the input subsystem that comprises all input neurons, output subsystem that consists of all output neurons, and it is possible to treat all neurons of the network or only all hidden neurons as its information processing subsystem [31]. In some cases, input and output neurons are regarded as one group of visible neurons.

Definition 3.2. *Infware* of an abstract automaton consists of objects processed by this automaton including input and output data.

Here are some examples.

The majority of abstract automata (computing devices) work with linear (i.e., one-dimensional) languages. Consequently, their infware consists of words in some alphabet.

Kolmogorov algorithms and storage modification machines work with arbitrary graphs [32,33]. Consequently, their infware consists of graphs.

There are also many practical algorithms that work with graphs (cf., for example, [34]). Consequently, their infware also consists of graphs.

Turing machines with two-dimensional tapes and two-dimensional cellular automata work with two-dimensional arrays of words. This means that their infware consists of two-dimensional arrays of words.

Turing machines with n-dimensional tapes and n-dimensional cellular automata work with n-dimensional arrays of words. This means that their infware consists of n-dimensional arrays of words.

Structural machines work with arbitrary structures [35,36].Consequently, their infware consists of arbitrary structures.

Note that such advanced abstract automata as structural machines can process not only data but also knowledge [37]. In this case, their infware consists of knowledge.

It is necessary to remark that the word *infoware*, which is used in the field of networks and computers, has a very different meaning in comparison with the term *infware*.

Definition 3.3. *Software* of an abstract automaton consists of texts, which control the functioning of this automaton.

Here are some examples.

Many kinds of algorithms and abstract automata, such as finite automata, pushdown automata, register machines, Kolmogorov algorithms, random access machines (RAM), and Turing machines, use systems of instructions, for example, in the form of transition rules, to control computational processes. Such systems of instructions constitute software of these automata and machines.

The system of weights, activation functions, threshold functions and output functions form software of neural networks. It is possible to treat these systems as algorithms although their form is different from traditional algorithms, which are described as sets of instructions.

In contrast to neural networks, software of the majority of algorithms and abstract automata consists of systems of instructions. These instructions or rules determine computational processes, which are controlled by algorithms and are going in these automata. All these classes of algorithms and abstract automata are unified by the comprehensive concept of an instruction machine.

Definition 3.4. a) An *instruction machine* or *instruction automaton M* is an automaton functioning of which is determined by a system of instructions (rules).

b) A *pure instruction machine* or *pure instruction automaton M* is an automaton functioning of which is determined only by a system of instructions (rules) and its input.

Note that the functioning of an instruction automaton is not necessarily uniquely determined by its system of instructions. For instance, its functioning can also depend on the states of its control device as in Turing machines. Besides, in nondeterministic instruction machines, e.g., in nondeterministic Turing machines, there are metarules of instruction selection, which can essentially influence its functioning [20,21].

At the same time, an important dynamic characteristic of the majority of abstract automata is their state. This brings us to another important class of automata.

Definition 3.5. a) A *state machine* or *state automaton M* has a control device and is an automaton functioning of which is determined by the states of its control device.

b) A *pure state machine* or *pure state automaton M* has a control device and is an automaton functioning of which is determined only by the states of its control device and its input.

Note that the control device of an automaton can coincide with the whole automaton. In this case, the functioning of the automaton is determined by its states. However, when the machine has a control device makes this automaton more flexible.

Often state machines (state automata) are also instruction machines (automata) with systems of instructions. However, implicitly any state machine (automaton) M is an instruction machine (automaton). Indeed, if we take descriptions of how the functioning of the machine (automaton) M depends on the state, we obtain instructions (rules) of its functioning.

We observe this situation in the case of finite automata. A finite automaton is a pure state machine (automaton) but its transition function (relation) makes it also an instruction machine (automaton).

Let us consider the structure of a state instruction machine (state instruction automaton) in a general case.

In a general case, a state instruction machine (state instruction automaton) M has three components:

- The *control device* C_M, which is a finite automaton and represents states of the machine (automaton) M;
- The *memory* W_M, which stores data;
- The *processor* P_M, which transforms (processes) information (data) from the input and the memory W_M.

The memory W_M consists of cells and connections between them. Each cell can be empty or contain a symbol from the alphabet A_M of the machine (automaton) M.

On each step of computation, the processor P_M observes one cell from the memory W_M at a time, and can change the symbol in this cell and go to another cell using connections in the memory W_M. These operations are performed according to the instructions R_M for the processor P_M. These instructions R_M can be stored in the processor P_M or in the memory W_M.

However, it is possible that an instruction machine consists of a single processor as its particular case—a finite automaton.

Example 3.1. A finite automaton G is an instruction machine, which has the following representation. Namely, a finite automaton (FA) G consists of three structures:

- The *linguistic structure* $L = (\Sigma, Q, \Omega)$ where Σ is a finite set of *input symbols*, Q is a finite *set of states*, and Ω is a finite set of *output symbols* of the automaton G;
- The *state structure* $S = (Q, q_0, F)$ where q_0 is an element from Q that is called the *start state* and F is a subset of Q that is called the set of *final* (in some cases, *accepting*) states of the automaton G;
- The *action structure*, which is traditionally called the *transition function* of G and has the following form

$$\delta: \Sigma \times Q \to Q \times \Omega$$

It can also be represented as two relation/functions—the state transition relation/function

$$\delta_1: \Sigma \times Q \to Q$$

and the output relation/function

$$\delta_2: \Sigma \times Q \to \Omega$$

Thus, a FA is a triad $G = (L, S, \delta)$.

The transition relation/function δ is portrayed by descriptions of separate transitions and each of these descriptions is an instruction (rule) for the automaton functioning.

Note that a finite automaton does not have a memory.

Example 3.2. A Turing machine T is also an instruction machine because its functioning is defined by a system of rules (instructions), which have the following form for a Turing machine with one head

$$qa \to pbQ$$

Here, a is the symbol, which is observed by the head of T and changed to the symbol b, and q is a state of the Turing machine, or more exactly, of its control device, which is changed in this operation to the state p, while Q is direction of the move of the head after performing the writing operation.

Example 3.3. An inductive Turing machine K is also an instruction machine because its functioning is defined by a system of rules (instructions), which are similar to rules of Turing machines [1,6].

There are numerous kinds of instruction machines with various types of instructions. However, it is possible to distinguish three classes of instructions:

- *Straightforward* or *prescriptive instructions* directly tell what is necessary to do.
- *Descriptive instructions* describe what result it is necessary to obtain.
- *Implicit instructions* have a form of data that can be interpreted as instructions.

Let us consider some examples.

Example 3.4. The descriptions of transitions of a finite automaton G are straightforward instructions.

Example 3.5. A function in functional programming is a descriptive instruction.

Example 3.6. Weights of artificial neurons in artificial neural networks are implicit instructions.

While all these examples are conventional models of computation, in the next section, we introduce and study more advanced models.

4. Structure of Triadic Automata and Machines

Triadic automata (machines) transform data (infware), instructions (software) and memory (hardware). Before describing their structure, we make an inventory of the types of triadic machines (triadic automata).

Definition 4.1. A triadic machine (automaton) is called:

- a *hardware modification machine* (*automaton*) if it transforms only infware and hardware,
- a *software modification* or *symmetric machine* (*automaton*) if it transforms only infware and software,
- a *transducer* if it transforms only infware and has input and output,
- a *generator* if it transforms only infware and has only output,
- an *acceptor* if it transforms only infware and only input,
- a *hardware expansion machine* (*automaton*) if it only expands its hardware,
- a *software expansion machine* (*automaton*) if it only expands its software, and
- a *symmetric expansion machine* (*automaton*) if it only expands its hardware and software.

Besides, there are different ways to perform hardware/software modifications. With respect to the source of modification, it is possible to consider three types of hardware/software modifications in an automaton (machine) M:

External modification is performed by another system.

Internal modification is performed by the automaton (machine) M.

Combined modification is performed by both the automaton (machine) M and another system.

What modifications are possible and permissible depends on the structure of a triadic machine or triadic automaton. Its mandatory components are input and output systems working together with one or more processors. Usually, input and output components are specific registers in the memory of the machine (automaton) [1]. At the same time, in neural networks, input and output are organized using specified neurons [31].

However, adding memory and other components to automata allows increasing their flexibility, interoperability and efficiency. These changes are reflected in the structure of triadic machines (automata), which have different types. Here we consider two types state and instruction triadic automata (machines).

Definition 4.2. A *triadic state machine* or *triadic state automaton A with memory* has seven core hardware components:

- The *control device* C_A, which is a finite automaton and represents states of the machine (automaton) A;
- The *data memory* W_A, which stores data and includes input and output registers;
- The *software memory* V_A, which stores software of the machine (automaton) A;
- The *data processor* P_M, which transforms (processes) information (data) from the memory W_M;
- The *software processor* D_M, which transforms (processes) software of A stored in the memory V_M;
- The *metaprocessor* P_A, which transforms (e.g., builds or deletes connections in) the hardware H_A and/or changes the control device C_A.

In the standard form, both memories consist of cells, which are connected by transition links. Processors have their programs of functioning, which constitute the software of the automaton.

In the same way as triadic state machines, triadic instruction machines constitute a special class of triadic machines. In a general case, it is possible that the functioning of a triadic instruction machine does depend on its state. However, we include the state system in the general description of triadic instruction machines because when the functioning of a triadic instruction machine does depend on its state, it is possible to treat this as a machine with only one state.

Definition 4.3. A triadic instruction machine or triadic instruction automaton H with memory has seven core hardware components:

- The *control device* C_H, which is a finite automaton and represents states of the machine (automaton) H;
- The *data memory* W_H, which stores data;
- The *instruction memory* V_H, which stores instructions;
- The *data processor* P_M, which transforms (processes) information (data) from the memory W_M;
- The *instruction processor* D_M, which transforms (processes) information (instructions) from the memory V_M;
- The *memory processor* P_W, which transforms (builds or deletes connections and/or cells in) the memory W_M;
- The *memory processor* P_V, which transforms (e.g., builds or deletes connections and/or cells in) the memory V_M.

Memory processors are hardware transformers and it is also possible to include a control device processor in the structure of a triadic instruction machine. This additional processor changes the control device C_A.

There are different classes of triadic instruction machines (automata).

Definition 4.4. Triadic instruction machines (automata) that:

- do not have processors that transform memory are called *symmetric instruction machines*,
- have only processor(s) that transform data are called *instruction machines*,
- have only processor(s) that transform instructions are called *translation machines* or *translators*,
- have only processor(s) that transform memory are called *construction machines* or *constructors*,
- do not have processors that transform data are called *constructors* (*construction machines*) *with translators*, and
- do not have processors that transform instructions are called *generative instruction machines*.

Machines from each class have their specific functions. For instance, construction machines (constructors) can be used to construct memory for other machines. This technique is employed in inductive Turing machines of the second and higher orders use inductive Turing machines of lower orders as their constructors [1, 6].

Besides, there are different methods to organize program formation with the help of computing/constructing agents. If the memory of the automaton has connections between any pair of cells, then the program can use these connections. Thus, it is possible to organize the inductive mode of computing by inductive computation (compilation) of the program for the main computation.

In the simplest approach called the *sequential strategy*, it is assumed that given some schema, for example, a description of the structure of the memory E of an inductive Turing machine M, an automaton A builds the program and places it in the memory E before the machine M starts its computation. When M is an inductive Turing machine of the first order, its constructor A is a Turing machine, which, for example, puts the names of the connections of the memory of M into instructions (rules) of M. When M is an inductive Turing machine of the second or higher order, its constructor A is also an inductive Turing machine—the order of which is less than the order of M

and which modifies instructions (rules) of *M*. For instance, the program of inductive Turing machines of the second order is constructed by Turing machines of the first order

According to another methodology, which is called the *concurrent strategy*, program formation by the automaton *A* and computations of the machine *M* go concurrently, that is, while the machine *M* computes, the automaton *A* constructs the program in the memory *E*.

It is also possible to use the *mixed strategy* when some parts of the program *E* are assembled before the machine *M* starts its computation, while other parts are formed parallel to the computing process of the machine *M*.

These three strategies determine three kinds of the constructed program (software):

- In the *static program* (*static software*) of the machine *M*, everything is constructed before *M* starts working.
- In the *growing program* (*growing software*) of the machine *M*, parts are constructed while *M* is working but no parts are deleted.
- In the *dynamic program* (*growing software*) of the machine *M*, when it necessary, some parts are constructed and when it necessary, some parts are deleted while *M* is working.

It is possible to use similar strategies for hardware modification. This approach determines three types of the constructed hardware of a triadic automaton/machine:

- In the *static hardware* of the machine *M*, everything is constructed before *M* starts working.
- In the *growing hardware* of the machine *M*, parts are constructed while *M* is working but no parts are deleted.
- In the *dynamic hardware* of the machine *M*, when it necessary, some parts are constructed and some parts are deleted while *M* is working.

Now, we can analyze the functioning of triadic automata and machines in more detail.

5. The Dynamics of Triadic Automata and Machines

To describe the dynamics of triadic automata and machines, we need some concepts from the theory of computation and automata.

There are different equivalence relations between automata, machines and algorithms. Three basic ones are determined by properties of computations [1, 38].

Definition 5.1. a) Two automata, machines or algorithms *A* and *B* are *operationally equivalent* if given the same input, they perform the same operations.

b) Two classes of automata, machines or algorithms **H** and **K** are *operationally equivalent* if each automaton in **H** is operationally equivalent to an automaton in **K** and vice versa.

For instance, a pushdown automaton, which does not use its stack, is operationally equivalent to a nondeterministic finite automaton. Consequently, the class of all pushdown automata, which do not use their stack, is operationally equivalent to the class of all nondeterministic finite automata.

However, operational equivalence does not completely characterize automata, machines and algorithms with respect to their functioning. To achieve this goal, we need a stronger equivalence.

Definition 5.2. a) Two operationally equivalent automata, machines or algorithms *A* and *B* are *strictly operationally equivalent* if for both of them the result is determined by the same rules.

b) Two classes of automata, machines or algorithms **H** and **K** are *strictly operationally equivalent* if each automaton in **H** is strictly operationally equivalent to an automaton in **K** and vice versa.

By definition, strictly operationally equivalent automata, machines or algorithms are operationally equivalent. However, properties of computation show that two automata (machines or algorithms) can be operationally equivalent but not strictly operationally equivalent. For instance, a Turing machine with one working tape, one input read-only tape, one output write-only tape and with three corresponding heads is operationally equivalent to a simple inductive Turing machine

[1]. However, these machines are not strictly operationally equivalent because their results are defined by different rules.

In a similar way, strictly operationally equivalent classes of automata, machines or algorithms are operationally equivalent. However, the previous example shows that two classes of automata (machines or algorithms) can be operationally equivalent but not strictly operationally equivalent.

In addition to two kinds of operational equivalence, there are other forms of equivalence of automata, machines and algorithms. In particular, it is known that automata can perform different operations but give the same result. This observation brings us to two more types of automata/machines equivalence.

Definition 5.3. a) Two classes of automata, machines or algorithms **H** and **K** are *functionally equivalent* if the automata, machines or algorithms from **H** and **K** compute the same class of functions.

b) Two automata, machines or algorithms A and B are *functionally equivalent* if the classes $\{A\}$ and $\{B\}$ are functionally equivalent.

For instance, the class of all Turing machines with one tape and the class of all one-dimensional cellular automata are functionally equivalent because they compute the same class of partially recursive functions [1].

Properties of automata machines and algorithms imply the following result.

Lemma 5.1. If the results of computations are defined in the same way for two (classes of) automata machines or algorithms, then their operational equivalence implies their functional equivalence.

In other words, strict operational equivalence implies functional equivalence.

However, when the results of computations are defined in a different way, two operationally equivalent (classes of) automata machines or algorithms can be not functionally equivalent. For instance, the class of all Turing machines with three tapes and heads and the class of all simple inductive Turing machines are operationally equivalent but they are not functionally equivalent because inductive Turing machines can compute much more functions than Turing machines [6].

Let us assume that all considered below automata (machines) work with words in some alphabet. Naturally, these automata (machines) compute some languages.

Definition 5.4. a) Two classes of automata, machines or algorithms **H** and **K** are *linguistically equivalent* if they compute (or accept) the same class of languages.

b) Two automata, machines or algorithms A and B are *linguistically equivalent* if the classes $\{A\}$ and $\{B\}$ are linguistically equivalent.

For instance, the class of all deterministic finite automata and the class of all nondeterministic finite automata are linguistically equivalent [1].

Properties of functions imply the following result.

Lemma 5.2. [38]. Functional equivalence of two (classes of) automata machines or algorithms implies their linguistic equivalence.

Lemmas 5.1 and 5.2 imply the following result.

Corollary 5.1. Strict operational equivalence implies linguistic equivalence.

However, operational equivalence implies neither functional nor linguistic equivalence. Indeed, when the results of computations are defined in a different way, two operationally equivalent (classes of) automata machines or algorithms can be not linguistically equivalent. For instance, the class of all Turing machines with three tapes and heads and the class of all simple inductive Turing machines are operationally equivalent but they are not linguistically equivalent because inductive Turing machines can compute much more languages than Turing machines [6].

All classes of automata, machines or algorithms studied and utilized in computer science and mathematics are usually divided into three substantial categories: subrecursive, subrecursive and superrecursive classes of automata, machines or algorithms [1]. Here we introduce a more detailed classification.

Definition 5.5. A class of automata, machines or algorithms **R** is called *linguistically* (*functionally*) *recursive* if it is linguistically (functionally) equivalent to the class of all Turing machines.

Example 5.1. The class of all Random Access Machines (RAM) is linguistically and functionally recursive [39].

Example 5.2. The class of all cellular automata is linguistically recursive [40].

Example 5.3. The class of all storage modification machines is linguistically and functionally recursive [33].

Example 5.4. The class of all Minsky machines is linguistically and functionally recursive [41].

Any language computed by some Turing machine is also computed by a universal Turing machine [42]. This gives us the following result.

Theorem 5.1. A class of automata or machines **H** is linguistically recursive if and only if it is linguistically equivalent to the class $\{U\}$ where U is a universal Turing machine.

Recursive classes of automata, machines or algorithms form a theoretical threshold for separating two basic classes—subrecursive and superrecursive automata, machines or algorithms. Namely, Turing machines are also used to define two more classes of automata, machines or algorithms.

Definition 5.6. A class of automata, machines or algorithms **W** is called *linguistically* (*functionally*) *subrecursive* if not all languages (functions) computable/acceptable in the class of all Turing machines are computable/acceptable in **W**.

Example 5.5. The class of all deterministic finite automata is linguistically and functionally subrecursive.

Example 5.6. The class of all pushdown automata is linguistically and functionally subrecursive.

Example 5.7. The class of all resource restricted Turing machines is linguistically and functionally subrecursive.

For a long time, it was believed that functionally recursive algorithms/automata in general and Turing machines in particular form the most computationally powerful class of algorithms/automata. In spite of this different classes of more powerful algorithms/automata have been constructed. They form one more basic class of algorithms/automata.

Definition 5.7. A class of automata, machines or algorithms **U** is called *linguistically* (*functionally*) *superrecursive* if not all languages (functions) computable/acceptable in **U** are computable/acceptable in the class of all Turing machines.

Example 5.8. The class of all inductive Turing machines is linguistically and functionally superrecursive [1, 6].

Example 5.9. The class of all periodic Turing machines is linguistically and functionally superrecursive [43].

Example 5.10. The class of all inductive cellular automata is linguistically and functionally superrecursive [44].

However, the possibility to compute a function (language) noncomputable by Turing machines does not guaranty that that in this class of algorithms/automata, all functions (languages) computable by Turing machines will be computable. In other words, it is possible that a linguistically and functionally superrecursive class of automata, machines or algorithms does not contain a linguistically and functionally recursive subclass of automata, machines or algorithms. That is why it is reasonable to consider one more important category of automata (machines) related to their functioning.

Definition 5.8. A superrecursive class of automata, machines or algorithms **U** is called *strictly linguistically* (*functionally*) *superrecursive* if all languages (functions) computable/acceptable in the class of all Turing machines are also computable/acceptable in **U**.

Example 5.11. The class of all simple inductive Turing machines is strictly linguistically and functionally superrecursive [1, 6].

Example 5.12. The class of all inductive cellular automata is strictly linguistically and functionally superrecursive [44]

Example 5.13. The class of all neural networks with real number weights is strictly linguistically and functionally superrecursive [45].

Example 5.14. The class of all general machines working with real numbers is strictly linguistically and functionally superrecursive [46, 47].

Lemma 5.3. Any strictly linguistically (functionally) superrecursive class of automata, machines or algorithms is linguistically (functionally) superrecursive.

The introduced stratification of classes of automata or machines into three groups allows obtaining similar classification of individual automata or machines.

Definition 5.9. a) An automaton (machine) A is *linguistically* (*functionally*) *recursive* if the class $\{A\}$ is linguistically (functionally) recursive.

b) An automaton (machine) A is *nominally linguistically* (*functionally*) *recursive* if it belongs to a class of linguistically (functionally) recursive automata (machines).

In other words, automaton (machine) A is linguistically (functionally) recursive if the class $\{A\}$ is linguistically (functionally) equivalent to the class of all Turing machines.

Example 5.15. A universal Turing machine is linguistically and functionally recursive.

Example 5.16. Any Turing machine is nominally linguistically and functionally recursive.

Note that any Turing machine is nominally linguistically and functionally recursive, but it is not always linguistically and functionally recursive.

Properties of universal Turing machines imply the following result.

Theorem 5.2. An automaton (machine) A is linguistically recursive if and only if it is linguistically equivalent to a universal Turing machine.

Definition 5.10. An automaton (machine) A is *linguistically* (*functionally*) *subrecursive* if the class $\{A\}$ is linguistically (functionally) subrecursive.

Example 5.17. Any finite automaton is linguistically and functionally subrecursive.

Note that any Turing machine is nominally linguistically and functionally recursive but in the majority of cases, it is linguistically and functionally subrecursive.

Proposition 5.1. A nominally linguistically (functionally) recursive automaton (machine) A is also nominally linguistically subrecursive if it is not linguistically (functionally) recursive and superrecursive.

Definition 5.11. a) An automaton (machine) A is *linguistically* (*functionally*) *superrecursive* if the class $\{A\}$ is linguistically (functionally) superrecursive.

b) An automaton (machine) A is *nominally linguistically* (*functionally*) *superrecursive* if it belongs to a class of linguistically (functionally) superrecursive automata (machines).

Example 5.18. A universal inductive Turing machine is linguistically and functionally superrecursive.

Note that any inductive Turing machine is nominally linguistically and functionally superrecursive but in the majority of cases, it is linguistically and functionally recursive or subrecursive. For instance, inductive Turing machines that do not use memory are linguistically and functionally subrecursive

This clearly shows the importance of the memory for the computing power of automata and machines. Structured memory is an important concept introduced in the theory of inductive Turing machines [48]. We remind that a structured memory E consists of cells and is structured by a system of relations and connections (ties) that organize memory functioning and provide connections between cells. This structuring delineates three components of the memory E: input registers, the working memory, and output registers. In a general case, cells may be of different types: binary cells store bits of information represented by symbols 1 and 0; byte cells store information represented by strings of eight binary digits; symbol cells store symbols of the alphabet(s) of the machine M and so on. Thus, the memory is a network of cells and it is possible to interpret these cells not only as containers of symbols but also as information processing systems, such as neurons in the brain of a

human being or computers in the World Wide Web or abstract computing devices in a grid automaton [49].

Definition 5.12. The software (hardware) of a triadic automaton is *recursive* if it and its transformation rules determine recursive algorithms.

Example 5.18. The software (hardware) of a Turing machine is recursive.

Let A be an instruction automaton (machine) with a structured memory.

Theorem 5.3. If the instructions (rules) of the automaton (machine) A are recursive, i.e., they define a recursive or subrecursive algorithm, but the structuring of its memory is superrecursive, then A can also be superrecursive.

The structuring of the automaton (machine) memory can be performed as hardware modification similar to the construction of inductive Turing machines of the second and higher orders. As inductive Turing machines of any order form a superrecursive class of automata, Theorem 5.3 give us the following result.

Corollary 5.2. It is possible to achieve superrecursivity by hardware modification.

Corollary 5.3. A triadic automaton with recursive software can be superrecursive.

An example of such an automaton is given in the studies of interactive hypercomputation [50].

Definition 5.13. Information processing, e.g., computation, is *recursive* if it can be realized by a recursive algorithm.

Properties of recursive computations imply the following result [1].

Theorem 5.4. A triadic automaton with recursive software and hardware and their parallel recursive data processing and software/hardware modification is recursive.

This result shows that parallel to data processing recursive software/hardware modification cannot increase computing power. However, even recursive rules of hardware modification can essentially improve efficiency of automata and machines decreasing their time complexity. Namely, results from [51] allow proving the following result.

Theorem 5.5. For any recursively computable language L (function f), there is a recursive hardware modification automaton A with a priory structured memory, which computes the language L (the function f), with linear time complexity.

As recursive hardware modification automata are also triadic automata, we have the following corollary.

Corollary 5.4. For any recursively computable language L (function f), there is a recursive triadic automaton A, which computes L (f) with linear time complexity.

Inductive Turing machines form an important class of automata formalizing scientific induction and inductive reasoning in general. That is why we can use inductive Turing machines for developing a new classification of automata, machines or algorithms. Here we do this for inductive Turing machines of the first order.

Definition 5.14. A class of automata, machines or algorithms **D** is called *linguistically* (*functionally*) *plainly inductive* if it is linguistically (functionally) equivalent to the class $\mathbf{IM}_1$ of all inductive Turing machines of the first order.

Example 5.19. The class of all simple inductive Turing machines is linguistically and functionally inductive [1, 6].

Example 5.20. The class of all periodic Turing machines is linguistically and functionally inductive [43].

Example 5.21. The class of all inductive cellular automata is linguistically and functionally inductive [44].

Properties of universal inductive Turing machines of the first order (cf. [1, 6]) give us the following result.

Theorem 5.6. A class of automata or machines **H** is linguistically plainly inductive if and only if it is linguistically equivalent to the class $\{W\}$ where W is a universal inductive Turing machine of the first order.

Definition 5.15. A class of automata, machines or algorithms **W** is called *linguistically* (*functionally*) *plainly superinductive* if not all languages (functions) computable/acceptable in **W** are computable/acceptable in the class of all inductive Turing machines of the first order.

Properties of universal inductive Turing machines of the second or higher orders (cf. [1, 6]) give us the following result.

Theorem 5.7. A class of automata or machines **H** is linguistically (functionally) plainly superinductive if it is linguistically (functionally) equivalent to the class $\mathbf{IM}_2$ of all inductive Turing machines of the second or higher order.

This brings us to the following problem.

Problem 1. Are there proper subclasses of the class $\mathbf{IM}_2$ of all inductive Turing machines of the second order, which are computationally weaker than $\mathbf{IM}_2$ but are still linguistically or functionally plainly superinductive?

6. Conclusion

We introduced concepts of triadic automata and machines. Relations between different types of triadic automata and machines and their properties were obtained. In addition, we further developed taxonomy of classes of automata and machines as well as of individual automata and machines. All this opens new directions for further research.

Inductive Turing machines form a powerful class of algorithms [6]. Thus, formalization and exploration of triadic inductive Turing machines is a motivating problem for future research. It is possible to study the same problem for periodic Turing machines [43], which are intrinsically related to inductive Turing machines.

Structural machines provide an extremely flexible model of computation [35, 36]. It would be interesting to introduce and study triadic structural machines.

It would also be interesting to formalize triadic cellular automata, triadic inductive cellular automata [44] and triadic neural networks and study their properties.

Traditional recursive and superrecursive models of automata and algorithms are used for the development of algorithmic information theory. Thus, an important direction for the future research is the development of triadic information theory based on triadic automata.

Funding: This research received no external funding

Acknowledgments: The author would like to express gratitude to Robert Prentner and another reviewer unknown to the author for their useful remarks.

Conflicts of Interest: The author declares no conflict of interest.

References

1. Burgin, M. *Superrecursive Algorithm*. Springer: New York, NY, USA, 2005.
2. Burgin, M. Inductive Turing Machines. In *Unconventional Computing—A volume in the Encyclopedia of Complexity and Systems Science*; Adamatzky, A., Ed.; Springer: Berlin/Heidelberg, Germany, 2018, pp. 675–688.
3. Dymond, P.W.; Cook, S.A. Hardware complexity and parallel computation. In Proceedings of the 21st Annual Symposium on Foundations of Computer Science, Syracuse, NY, USA, 13–15 October 1980; pp. 360–372.
4. Cook, S.A. Towards a complexity theory of parallel computation. *L'Enseignement Math.* **1981**, *XXVII*, 99–124.
5. Dymond, P.W. On nondeterminism in parallel computation. *Theor. Comput. Sci.* **1986**, *47*, 111–120.
6. Burgin, M. Nonlinear Phenomena in Spaces of Algorithms. *Int. J. Comput. Math.* **2003**, *80*, 1449–1476.
7. Kleene, S. Constructive and Non-constructive Operations. In Proceedings of the International Congress of Mathematicians, Edinburg, UK, 14–21 August 1958; Cambridge University Press: Cambridge, UK, 1960.
8. Burgin, M. Reflexive Calculi and Logic of Expert Systems. In *Creative Processes Modeling by Means of Knowledge Bases*; Institute of Mathematics: Sofia, Bulgaria, 1992; pp. 139–160. (In Russian)

9. Burgin, M. Reflexive Turing Machines and Calculi. *Vychislitelnyye Syst. (Log. Methods Comput. Sci.)* **1993**, *148*, 94–116, 175–176.
10. Schroeder, M.J. Dualism of Selective and Structural Manifestations of Information in Modelling of Information Dynamics. In *Computing Nature*; SAPERE 7; Springer: Berlin, Germany, 2013; pp. 125–137.
11. Schroeder, M.J. From Proactive to Interactive Theory of Computation. In Proceedings of the 6th AISB Symposium on Computing and Philosophy: The Scandal of Computation—What is Computation? Exeter, UK, 2–5 April 2013; pp. 47–51.
12. Burgin, M. Processing information by symmetric inductive machines. Presented at theIS4SI Summit Berkeley 2019—Where is the I in AI and the Meaning in Information, Berkeley, CA, USA, 2–7 June 2019.
13. Smith, B.C. Procedural Reflection in Programming Languages. Ph.D. Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, USA, 1982.
14. Demers, F.-N.; Malenfant, J. Reflection in logic, functional and object-oriented programming: A Short Comparative Study. In Proceedings of the IJCAI'95 Workshop on Reflection and Metalevel Architectures and their Applications in AI, Montreal, Canada, 20–25 August, 1995, pp. 29–38.
15. Chlipala, A. Ur: statically-typed metaprogramming with type-level record computation. In Proceedings of the ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation (PLDI'10), Phoenix, AZ, USA, 22–28 June 2010; Volume 45, pp. 122–133.
16. Burgin, M. *Structural Reality*; Nova Science Publishers: New York, NY, USA, 2012.
17. Mandler, J.M. *Stories, Scripts, and Scenes: Aspects of Schema Theory*; Lawrence Erlbaum Associates: Hillsdale, NJ, USA, 1984.
18. Arbib, M.A. *Modules, Brains and Schemas*, *Formal Methods*; LNCS 3393; Springer: Berlin/Heidelberg, Germany, 2005, pp. 153–166.
19. Burgin, M. Mathematical Schema Theory for Modeling in Business and Industry. In Proceedings of the 2006 Spring Simulation Multi Conference (SpringSim'06), Huntsville, AL, USA, 2–6 April 2006; pp. 229–234.
20. Burgin, M.**;** Debnath, N. Reusability as Design of Second-Level Algorithms. In Proceedings of the ISCA 25th International Conference "Computers and their Applications" (CATA-2010), ISCA, Honolulu, HI, USA, 22–24 October 2010; pp. 147–152.
21. Burgin, M.; Gupta, B. Second-level Algorithms, Superrecursivity, and Recovery Problem in Distributed Systems. *Theory Comput. Syst.* **2012**, *50*, 694–705.
22. Dehaene, S.; Changeux, J.P. Development of elementary numerical abilities: A neuronal model. *J. Cogn. Neurosci*. **1993**, *5*, 390–407.
23. Dehaene, S. *The Number Sense*: *How the Mind Creates Mathematics*; Oxford University Press: New York, NY, USA, 1997.
24. Schroeder, M.J. Computing with Nature. *Proceedings* **2017**, *1*, 178.
25. Dodig-Crnkovic, G.; Burgin, M. *Information and Computation*; World Scientific: New York, NY, USA; 2011.
26. Zuse, K. *Rechnender Raum*; Friedrich Vieweg&Sohn: Braunschweig, Germany, 1969.
27. Fredkin, E. Digital Mechanics. *Phys. D* **1990**, *45*, 254–270.
28. Dodig-Crnkovic, G. Significance of Models of Computation from Turing Model to Natural Computation. *Minds Mach.* **2011**, *21*, 301–322.
29. Burgin, M.; Dodig-Crnkovic, G. Information and Computation—Omnipresent and Pervasive. In *Information and Computation*; World Scientific: New York, NY, USA, 2011; pp. vii–xxxii.
30. Shiva, S.G. *Advanced Computer Architectures*; CRC Press: Boca Raton, FL, USA, 2005
31. Haykin, S. *Neural Networks: A Comprehensive Foundation*; Macmillan: New York, NY, USA, 1994.
32. Kolmogorov, A.N. On the concept of algorithm, *Uspehi Mat. Nauk*. **1953**, *8*, 175–176
33. Shönhage, A. Storage Modification Machines. *SIAM J. Comput.* **1980**, *9*, 490–508.
34. Kleinberg, J.; Tardos, E. *Algorithm Design*; Pearson-Addison-Wesley: Boston, MA, USA, 2006.
35. Burgin, M.; Adamatzky, A. Structural machines and slime mold computation, *Int. J. Gen. Syst.* **2017**, *45*, 201–224.
36. Burgin, M.; Adamatzky, A. Structural Machines as a Mathematical Model of Biological and Chemical Computers. *Theory Appl. Math. Comput. Sci.* **2017**, *7*, 1–30
37. Burgin, M.; Mikkilineni, R.; Mittal, S. Knowledge processing as structure transformation. *Proceedings* **2017**, *1*, 212. doi:10.3390/IS4SI-2017-03988.
38. Burgin, M. *Measuring Power of Algorithms*, *Computer Programs*, *and Information Automata*; Nova Science Publishers: New York, NY, USA, 2010.

39. Shepherdson, J.C.; Sturgis, H.E. Computability of Recursive Functions. *J. ACM* **1963**, *10*, 217–255.
40. von Neumann, J. *Theory of Self-Reproducing Automata*; University of Illinois Lectures on the Theory and Organization of Complicated Automata, Edited and completed by Arthur W. Burks. University of Illinois Press: Urbana, IL, USA, 1949.
41. Minsky, M. *Computation: Finite and Infinite Machines*; Prentice-Hall: New York, NY, USA, 1967.
42. Hopcroft, J.E.; Motwani, R.; Ullman, J.D. *Introduction to Automata Theory, Languages, and Computation*; Addison Wesley: Boston, MA, USA, 2007.
43. Burgin, M. Periodic Turing Machines. *J. Comput. Technol. Appl. (JoCTA)* **2014**, *5*, 6–18.
44. Burgin, M. Inductive Cellular Automata. *Int. J. Data Struct. Algorithms* **2015**, *1*, 1–9
45. Siegelman, H.T. *Neural Networks and Analog Computation*: *Beyond the Turing Limit*; Birkhauser: Berlin, Germany, 1999.
46. Blum, L.; Shub, M.; Smale, S. On a theory of computation and complexity over the real numbers: NP-completeness, recursive functions and universal machines. *Bull. Am. Math. Soc*. **1989**, *21*, 1.
47. Blum, L.; Cucker, F.; Shub, M.; Smale, S. *Complexity and Real Computation*; Springer: New York, NY, USA, 1998.
48. Burgin, M.S. Inductive Turing Machines. *Not. Acad. Sci. USSR* **1983**, *270*, 1289–1293.
49. Burgin, M. Cluster Computers and Grid Automata. In Proceedings of the ISCA 17th International Conference "Computers and their Applications", International Society for Computers and their Applications, Honolulu, HI, USA, 13–15 August 2003; pp. 106–109.
50. Burgin, M. Interactive Hypercomputation. In Proceedings of the 2007 International Conference on Foundations of Computer Science (FCS'07), Las Vegas, NV, USA, 25–28 June 2007; CSREA Press: Las Vegas, NV, USA, 2007; pp.328–333.
51. Burgin, M. Super-recursive Algorithms as a Tool for High Performance Computing. In Proceedings of the High Performance Computing Symposium, San Diego, CA, USA, 12–15 April 1999; pp. 224–228.