Theory and Applications of Mathematics & Computer Science

(ISSN 2067-2764, EISSN 2247-6202)
http://www.uav.ro/applications/se/journal/index.php/tamcs

Theory and Applications of Mathematics & Computer Science 5 (1) (2015) 71–93

Properties of Stabilizing Computations

Mark Burgin[a]

[a]*University of California, Los Angeles 405 Hilgard Ave. Los Angeles, CA 90095*

Abstract

Models play an important role in the development of computer science and information technology applications. Turing machine is one of the most popular model of computing devices and computations. This model, or more exactly, a family of models, provides means for exploration of capabilities of information technology. However, a Turing machine stops after giving a result. In contrast to this, computers, networks and their software, such as an operating system, very often work without stopping but give various results. There are different modes of such functioning and Turing machines do not provide adequate models for these processes. One of the closest to halting computation is stabilizing computation when the output has to stabilize in order to become the result of a computational process. Such stabilizing computations are modeled by inductive Turing machines. In comparison with Turing machines, inductive Turing machines represent the next step in the development of computer science providing better models for contemporary computers and computer networks. At the same time, inductive Turing machines reflect pivotal traits of stabilizing computational processes. In this paper, we study relations between different modes of inductive Turing machines functioning. In particular, it is demonstrated that acceptation by output stabilizing and acceptation by state stabilizing are linguistically equivalent.

Keywords: computation, stability, Turing machine, inductive Turing machine, acceptation, mode of computation, equivalence.
2010 MSC: 68Q05.
2012 CCS: theory of computation, models of computation.

1. Introduction

Computer science studies computations by means of theoretical models. One of the most popular theoretical models of computation is Turing machine. It is central in computer science and in many applications, especially, when it is necessary to prove impossibility of an algorithmic solution to a problem (Rogers, 1987). The pivotal feature of a Turing machine is the necessity to stop after giving a result because all subsequent machine operations become superfluous.

However, computers, networks and their software, such as an operating system, very often work without stopping but give various results. There are different modes of such functioning and Turing machines do not provide adequate models for these processes (Burgin, 2005*a*). Stabilizing computation is one of the closest to halting computation computational modes when the output has to stabilize in order to become the result of a computational process. Such stabilizing computations

are efficiently modeled by inductive Turing machines (Burgin & Debnath, 2004, 2005; Burgin, 2005*a*, 2006; Burgin & Gupta, 2012).

In comparison with Turing machines, inductive Turing machines represent the next step in the development of computer science providing better modeling tools for contemporary computers and computer networks (Burgin, 2005*a*). In particular, even simple inductive Turing machines and other inductive Turing machines of the first order can solve the Halting Problem for Turing machines, while inductive Turing machines of higher orders can generate and decide the whole arithmetical hierarchy as it is proved in (Burgin, 2003). Even more, unrestricted inductive Turing machines with a structured memory have the same computing power as Turing machines with oracles (Burgin, 2005*a*). In addition, inductive Turing machines allow decreasing time of computations (Burgin, 1999). Being more powerful, inductive Turing machines allow essential reduction of Kolmogorov (algorithmic) complexity of finite objects (Burgin, 2004), as well as algorithmic complexity of mathematical and computational problems (Burgin, 2010*a*). It is also important that in contrast to Turing machines, which can work only with words (Turing machines with one one-dimensional tape), with finite systems of words (Turing machines with several one-dimensional tapes) and with arrays (Turing machines with multidimensional tapes), inductive Turing machines can work not only with finite and infinite words, systems of words and multidimensional arrays but also with more sophisticated data structures, such as graphs, functions, hierarchical structures and chains of named sets or named data.

Inductive Turing machines have found applications in algorithmic information theory and complexity studies (Burgin, 2004, 2007, 2010*a*), software testing (Burgin & Debnath, 2009; Burgin *et al.*, 2009), high performance computing (Burgin, 1999), machine learning (Burgin & Klinger, 2004), software engineering (Burgin & Debnath, 2004, 2005), computer networks (Burgin, 2006; Burgin & Gupta, 2012) and evolutionary computations (Burgin & Eberbach, 2008, 2009*b*,*a*, 2010, 2012). For instance, inductive Turing machines can perform all types of machine learning - TxtEx-learning, TxtFin-learning, TxtBC-learning, and TxtEx*-learning, (Beros, 2013). While the traditional approach to machine learning models learning processes using functions, e.g., limit partial recursive functions (Gold, 1967), inductive Turing machines are automata, which can compute values of the modeling functions.

An important area of tentative application of inductive Turing machines and other super-recursive algorithms is software development and maintenance. As Călinescu, et al, (Calinescu *et al.*, 2013) write, modern software systems are often complex, inevitably distributed, and operate in heterogeneous and highly dynamic environments. Examples of such systems include those from the service-oriented, cloud computing, and pervasive computing domains. In these domains, continuous change is the norm and therefore the software must also change accordingly. In many cases, the software is required to self-react by adapting its behavior dynamically, in order to ensure required levels of service quality in changing environments. As a result, conventional recursive algorithms, such as Turing machines, cannot provide efficient means for modeling software functioning and behavior. This can be achieved only by utilization of inductive Turing machines and other super-recursive algorithms. Thus, better knowledge of inductive Turing machines properties and regularities of their behavior allows their better utilization and application of these models of computation and computer systems.

The goal of this paper is to study inductive Turing machines as models of real computing

devices, functioning of which often results in stabilizing computations. Note that stability is an important property of a computing device, as well as of computational processes. By definition, inductive Turing machines give results if and only if their computational process stabilizes (Burgin, 2005*a*).

Each real computing device, e.g., a computer, has three components: an input device (devices), an output device (devices) and a processor or a multiprocessor, which consists of several processors. Consequently, there are three modes of component functioning: the input mode, output mode and processing mode. Together they form the functioning mode of the computing device. For instance, the processing mode of an automaton can be acceptation (of the input), decision (about the input) or computation (of the final result). The output mode of a pushdown automaton can be acceptance by final state or acceptance by empty stack (Hopcroft *et al.*, 2001). The output mode of a Turing machine can be acceptance by final state or acceptance by halting (Hopcroft *et al.*, 2001). An inductive Turing machine can process information in a recursive mode or in the inductive mode (Burgin, 2005*a*). Computer scientists are usually interested in equivalence of different computational modes as it allows them to use the most appropriate mode for solving a given problem without loss of generality.

There are different types of equivalence: linguistic equivalence, functional equivalence, process equivalence, etc. (cf. (Burgin, 2010*b*). Here we study the classical case of equivalence called linguistic equivalence.

We remind that two automata (computing devices) are linguistically equivalent if they have the same language (Burgin, 2010*b*). Note that it may be linguistic equivalence with respect to computation when the language of the automaton A is the language computed by A or it may be linguistic equivalence with respect to acceptation when the language of the automaton A is the language accepted by A.

We remind that two classes of automata are linguistically equivalent if they have the same classes of languages and two modes of functioning are linguistically equivalent if the classes of automata working in these modes are linguistically equivalent (Burgin, 2010*b*). As separate automata, classes of automata may be linguistically equivalent with respect to computation or with respect to acceptation. One of the basic results of the theory of pushdown automata is the statement that acceptance by final state is linguistically equivalent to acceptance by empty stack ((Hopcroft *et al.*, 2001) Section 6.2).

One of the basic results of the theory of Turing machines and recursive computations is the statement that acceptance by final state is linguistically equivalent to acceptance by halting ((Burgin, 2005*a*), Chapter 2).

The goal of this paper is to analyze different modes of inductive Turing machine functioning, finding whether similar results are true for these modes. Note that inductive Turing machines have much more modes of functioning than Turing machines. In Section 2, we remind some basic concepts and constructions from the theory of inductive Turing machines and stabilizing computations. In Section 3, we demonstrate that some key modes of inductive Turing machine functioning are linguistically equivalent.

2. Inductive Turing machines as models of stabilizing computations

To understand how inductive Turing machines model stabilizing computations, we need to know the hardware structure and characteristics of inductive Turing machine functioning in general and of the simple inductive Turing machine functioning, in particular, making the emphasis on work with finite words in some alphabet (Burgin, 2005*a*).

An inductive Turing machine M hardware consists of three abstract devices: a *control device A*, which is a finite automaton and controls performance of *M*; a *processor* or *operating device H*, which corresponds to one or several *heads* of a conventional Turing machine; and the *memory E*, which corresponds to the *tape* or tapes of a conventional Turing machine. The memory *E* of the simplest inductive Turing machine consists of three linear tapes, and the operating device consists of three heads, each of which is the same as the head of a Turing machine and works with the corresponding tapes. Such machines are called *simple inductive Turing machines* (Burgin, 2005*a*).

The *control device A* is a finite automaton. It controls and regulates processes and parameters of the machine *M*: the state of the whole machine *M*, the processing of information by *H*, and the storage of information in the memory *E*.

The *memory E* of a general inductive Turing machines is divided into different but, as a rule, uniform cells. It is structured by a system of relations that organize memory as well-structured system and provide connections or ties between cells. In particular, *input* registers, the *working* memory, and *output* registers of *M* are discerned. Connections between cells form an additional structure *K* of *E*. Each cell can contain a symbol from an alphabet of the languages of the machine *M* or it can be empty.

In a general case, cells may be of different types. Different types of cells may be used for storing different kinds of data. For example, binary cells, which have type B, store bits of information represented by symbols 1 and 0. Byte cells (type BT) store information represented by strings of eight binary digits. Symbol cells (type SB) store symbols of the alphabet(s) of the machine M. Cells in conventional Turing machines have SB type. Natural number cells, which have type NN, are used in random access machines. Cells in the memory of quantum computers (type QB) store q-bits or quantum bits. Cells of the tape(s) of real-number Turing machines (Burgin, 2005*a*) have type RN and store real numbers. When different kinds of devices are combined into one, this new complex device may have several types of memory cells. In addition, different types of cells facilitate modeling the brain neuron structure by inductive Turing machines.

The *processor H* performs information processing in *M*. However, in comparison to computers, *H* performs very simple operations. When *H* consists of one unit, it can change a symbol in the cell that is observed by *H*, and go from this cell to another using a connection from *K*. It is possible that the processor *H* consists of several processing units similar to heads of a multihead Turing machine. This allows one to model various real and abstract computing systems: multiprocessor computers; Turing machines with several tapes; networks, grids and clusters of computers; cellular automata; neural networks; and systolic arrays.

The *software R* of the inductive Turing machine *M* is also a program that consists of simple rules:

$$q_h a_i \to a_j q_k c \tag{2.1}$$

Here q_h and q_k are states of A, a_i and a_j are symbols of the alphabet of M, and c is a type of connection in the memory E. The rule (2.1) means that if the state of the control device A of M is q_h and the processor H observes in the cell the symbol a_j, then the state of A becomes q_k, while the processor H writes the symbol a_j in the cell where it is situated and moves to the next cell by a connection of the type c. Each rule directs one step of computation of the inductive Turing machine M. Rules of the inductive Turing machine M define the transition function of M and describe changes of A, H, and E. Consequently, these rules also determine the transition functions of A, H, and E.

These rules cover only synchronous parallelism of computation. However, it is possible to consider inductive Turing machines of any order in which their processor can perform computations in the concurrent mode. Besides, it is also possible to consider inductive Turing machines with several processors. These models of computation are studied elsewhere.

A general step of the machine M has the following form. At the beginning, the processor H observes some cell with a symbol a_i (it may be Λ as the symbol of an empty cell) and the control device A is in some state q_h. Then the control device A (and/or the processor H) chooses from the system R of rules a rule r with the left part equal to $q_h a_i$ and performs the operation prescribed by this rule. If there is no rule in R with such a left part, the machine M stops functioning. If there are several rules with the same left part, M works as a nondeterministic Turing machine, performing all possible operations. When A comes to one of the final states from F, the machine M also stops functioning. In all other cases, it continues operation without stopping.

In the output stabilizing mode, M gives the result when M halts and its control device A is in a final state from F, or when M never stops but at some step of the computation the content of the output register becomes fixed and does not change (cf. Definition 3.4). The computed result of M is the word that is written in the output register of M. In all other cases, M does not give the result (cf. Definition 3.6).

Now let us build a constructive hierarchy of inductive Turing machines.

The memory E is called *recursive* if all relations that define its structure are recursive. Here recursive means that there are Turing machines that decide or build the structured memory (Burgin, 2005*a*). There are different techniques to organize this process. The simplest approach assumes that given some data, e.g., a description of the structure of E, a Turing machine T builds all connections in the memory E before the machine M starts its computation. According to another methodology, memory construction by the machine T and computations of the machine M go concurrently, while the machine M computes, the machine T constructs connections in the memory E. It is also possible to consider a situation when some connections in the memory E are assembled before the machine M starts its computation, while other connections are formed parallel to the computing process of the machine M.

Besides, it is possible to consider a schema when the machine T is separate from the machine M, while another construction adopts the machine T as a part of the machine M.

Inductive Turing machines with recursive memory are called *inductive Turing machines of the first order*.

While in inductive Turing machines of the first order, the memory is constructed by Turing machines or other recursive algorithms, it is possible to use inductive Turing machines for memory construction for other inductive Turing machines. This brings us to the concept of inductive Turing

machines of higher orders. For instance, in inductive Turing machines of the second order, the memory is constructed by Turing machines of the first order.

In general, we have the following definitions.

The memory E is called *n-inductive* if its structure is constructed by an inductive Turing machine of the order n. Inductive Turing machines with n-inductive memory are called *inductive Turing machines of the order n + 1*. Namely, in inductive Turing machines of order n, the memory is constructed by Turing machines of order *n - 1*.

We denote the class of all inductive Turing machines of the order n by $\mathbf{IT_n}$ and take $\mathbf{IT} = \bigcup_{\mathbf{n=1}}^{\infty} \mathbf{IT_n}$.

In such a way, we build a constructive hierarchy of inductive Turing machines. Algorithmic problems solved by these machines form a superrecursive hierarchy of algorithmic problems (Burgin, 2005*b*).

A simple inductive Turing machine has the same structure and the same rules (instructions) as a conventional Turing machine with three heads and three linear tapes: the input tape, output tape and working tape. The input tape is a read-only tape and the output tape is a write-only tape.

Computation or acceptation of a simple inductive Turing machine M consists of two stages. At first, M rewrites the input word from the input tape to the working tape. Then M starts working with this word in the working tape, writing something to the output tape from time to time.

Thus, the rules of a simple inductive Turing machine have the form

$$q_h(a_{i1}, a_{i2}, a_{i3}) \to (a_{j1}, a_{j2}, a_{j3})q_k(T_1, T_2, T_3) \tag{2.2}$$

The meaning of symbols in formula (2.2) is similar to notations used for Turing machines. Namely, we have:

- q_h and q_k are states of the control device A;
- $a_{i1}, a_{i2}, a_{i3}, a_{j1}, a_{j2}$ and a_{j3} are symbols from the alphabet of M;
- each of the symbols T_1, T_2 and T_3 is equal either to L, which denotes the transition of the head to the left adjacent cell or to R, which denotes the transition of the head to the right adjacent cell, or to N, which denotes absence of a head transition.

The rule (2.2) means that if the state of the control device A of M is q_h and the head h_t observes in the cell of the tape t the symbol a_{it} , then the state of A becomes q_k , while the head h_t writes the symbol a_j in the cell where it is situated and moves to the direction indicated by $T_t(t = 1, 2, 3)$. Each rule directs one step of computation of the inductive Turing machine M.

It means that moves of a simple inductive Turing machine are the same as moves of a Turing machine with three tapes. The difference is in output. A Turing machine produces a result only when it halts. The result is a word on the output tape. A simple inductive Turing machine is also doing this but in addition, it produces its results without stopping. It is possible that in the sequence of computations after some step, the word on the output tape is not changing, while the simple inductive Turing machine continues working. This word, which is not changing, is the result of the machine that works in the output stabilizing computing mode. Thus, the simple

inductive Turing machine does not halt, producing a result after a finite number of computing operations.

Because here we consider inductive Turing machines that work only with finite words and processing of the input word starts only after it is rewritten into the working tape, it is possible to assume that there is no input tape, the initial word is written in the working tape and the rules have the form

$$q_h(a_{i2}, a_{i3}) \to (a_{j2}, a_{j3})q_k(T_2, T_3) \tag{2.3}$$

Here a_{i2} and a_{j2} are symbols in the working tape, a_{i3} and a_{j3} are symbols in the output tape, the symbol T_2 directs the move of the head h_2, while the symbol T_3 directs the move of the head h_3.

Besides, it is also possible to assume that the output written in the output tape does not influence operations of the inductive Turing machine because the output tape is the write-only tape.

3. Comparing results of stabilizing computations

In this section, we study functioning of inductive Turing machines of the first order with one linearly ordered output register and one linearly ordered input register (Burgin, 2005*a*). We also assume that these inductive Turing machines work with finite words in some alphabet. The main emphasis here is on simple inductive Turing machines. Note that in general, inductive Turing machines can work not only with finite and infinite words but also with multidimensional arrays, graphs and even more sophisticated data structures.

In the previous section, we considered only the output stabilizing computing mode of inductive Turing machines. However, it is possible to use other modes of inductive Turing machine functioning, for example, the state stabilizing mode. This is similar to the functioning modes of finite automata that work with infinite words, (Burks & Wright, 1953; Büchi, 1960; Thomas, 1990; Chadha *et al.*, 2009).

The simplest modes of inductive Turing machine functioning are acceptation by halting and computation by halting. Namely, we have:

Definition 3.1. a) An inductive Turing machine M accepts the input word *by halting* if after some number of steps, the machine M stops.

b) The set $L_{ht}(M)$ of all words accepted by halting of an inductive Turing machine M is called the *halting accepted language* of the machine M.

This is the standard mode for Turing machines (Hopcroft *et al.*, 2001; Burgin, 2005*a*).
Computation by halting is defined in a similar way.

Definition 3.2. a) An inductive Turing machine M computes a word w *by halting* if after some number of steps, the machine M stops and when this happens, w is the word in output tape.

b) The set $L^{ht}(M)$ of all words computed by halting of an inductive Turing machine M is called the *halting computed language* of the machine M.

In the theory of inductive Turing machines, it is proved that when an inductive Turing machine M gives the result by halting, i.e., accepts a word by halting or computes a word by halting, M is linguistically equivalent to a Turing machine, i.e., the language produced (accepted or computed) by M can be produced by some Turing machine (Burgin, 2005*a*). In the theory of Turing machines, it is proved that acceptation by halting is linguistically equivalent to computation by halting. This gives us the following result.

Proposition 3.1. *Computation by halting is linguistically equivalent to acceptation by halting in the class of all inductive Turing machines of the first order.*

Thus, inductive Turing machine M of the first order can compute and accept all recursively enumerable languages and only these languages.

Now we will study more productive modes of inductive Turing machine functioning when machines are able to compute or accept languages that are not recursively enumerable.

Let us consider an inductive Turing machine M. In the set Q of states of the machine M, several subsets $F_1, \ldots, F_k$ are selected and called the *final groups of states* of the inductive Turing machine M.

Definition 3.3. An inductive Turing machine M gives a *result by state stabilizing* if after some number of steps, the state of the machine M always remains in the same final group of states.

Note that traditionally states of the control device of a Turing machine or of an inductive Turing machine are treated as states of the whole machine (Burgin, 2005*a*; Hopcroft *et al.*, 2001; Sipser, 1996). We follow this tradition.

It is possible that one or several final groups consist of a single state. Then stabilization in such a group means that the state of the machine M always stops changing after some number of steps.

When the processing mode is acceptation, the result is acceptation of the input word. We formalize this situation by the following definition.

Definition 3.4. a) An inductive Turing machine M accepts the input word *by state stabilizing* if after some number of steps, the state of the machine M always remains in the same final group of states.

b) The set $L_{ht}(M)$ of all words accepted by state stabilizing of an inductive Turing machine M is called the *state stabilizing accepted language* of the machine M.

It is natural to consider the state stabilizing language $L_{ot}(M)$ of the machine M as the result of M working in the acceptation mode.

We denote by $\mathbf{L}_{st}$(ITM1) the set of all state stabilizing accepted languages of inductive Turing machines of the first order and by $\mathbf{L}_{st}$(SITM) the set of all state stabilizing accepted languages of simple inductive Turing machines.

In the computing mode, the result is defined in a different way, which is formalized by the following definition.

Definition 3.5. a) An inductive Turing machine M computes the input word w *by state stabilizing* if after some number of steps, the state of the machine M always remains in the same final group of states and w is the first word in the output tape when the state stabilization process starts. This output w is the final result of the machine M.

b) The set $L^{st}(M)$ of all words computed by state stabilizing of an inductive Turing machine M is called the *state stabilizing computed language* of the machine M.

It is natural to consider the state stabilizing language $L_{ot}(M)$ of the machine M as the result of M working in the acceptation mode.

We denote by $\mathbf{L}^{st}$(ITM1) the set of all state stabilizing computed languages of inductive Turing machines of the first order and by $\mathbf{L}^{st}$(SITM) the set of all state stabilizing computed languages of simple inductive Turing machines.

Definition 3.6. An inductive Turing machine M gives a *result by output stabilizing* if after some number of steps the output of the machine M stops changing. This output is the final result of the machine M.

When the processing mode is acceptation, the result is acceptation of the input word. Namely, we have the following concept.

Definition 3.7. a) An inductive Turing machine M accepts the input word *by output stabilizing* if after some number of steps, the output of the machine M stops changing. This output is the final result of the machine M.

b) The set $L_{ot}(M)$ of all words accepted by output stabilizing of an inductive Turing machine M is called the *output stabilizing accepted language* of the machine M.

It is natural to consider the output stabilizing language $L_{ot}(M)$ of the machine M as the result of M working in the acceptation mode.

We denote by $\mathbf{L}_{ot}$(ITM1) the set of all output stabilizing accepted languages of inductive Turing machines of the first order and by $\mathbf{L}_{ot}$(SITM)the set of all state stabilizing accepted languages of simple inductive Turing machines.

Definition 3.8. a) An inductive Turing machine M computes the input word *by output stabilizing* if after some number of steps, the output of the machine M stops changing. This output is the final result of the machine M.

b) The set $L^{ot}(M)$ of all words computed by output stabilizing of an inductive Turing machine M is called the *output stabilizing computed language* of the machine M.

It is natural to consider the state stabilizing language $L^{ot}(M)$ of the machine M as the result of M working in the computation mode.

We denote by $\mathbf{L}^{ot}$(ITM1) the set of all state stabilizing computed languages of inductive Turing machines of the first order and by $\mathbf{L}^{ot}$(SITM) the set of all state stabilizing computed languages of simple inductive Turing machines.

Let us consider more restrictive modes of inductive Turing machine functioning.

Definition 3.9. An inductive Turing machine M gives a *result by bistabilizing* if after some number of steps the output of the machine M stops changing, while the state of M remains in the same final group of states.

This output is the final result of the machine M.

When the processing mode is acceptation, the result is acceptation of the input word. Namely, we have the following concept.

Definition 3.10. a) An inductive Turing machine M accepts the input word *by bistabilizing* if after some number of steps, the output of the machine M stops changing, while the state of M remains in the same final group of states.

b) The set $L_{bt}(M)$ of all words accepted by bistabilizing of an inductive Turing machine M is called the *bistabilizing accepted language* of the machine M.

It is natural to consider the output stabilizing language $L_{bt}(M)$ of the machine M as the result of M working in the acceptation mode.

We denote by $\mathbf{L}_{bt}$(ITM1) the set of all bistabilizing accepted languages of inductive Turing machines of the first order and by $\mathbf{L}_{bt}$(SITM) the set of all bistabilizing accepted languages of simple inductive Turing machines.

Definition 3.11. a) An inductive Turing machine M computes the input word *by bistabilizing* if after some number of steps, the output of the machine M stops changing, while the state of M remains in the same final group of states.
The output that stopped changing is the final result of the machine M.

b) The set $L^{bt}(M)$ of all words computed by bistabilizing of an inductive Turing machine M is called the *bistabilizing computed language* of the machine M.

It is natural to consider the bistabilizing language $L^{bt}(M)$ of the machine M as the result of M working in the computation mode.

We denote by $\mathbf{L}^{bt}$(ITM1) the set of all bistabilizing computed computed languages of inductive Turing machines of the first order and by $\mathbf{L}^{bt}$(SITM) the set of all bistabilizing computed languages of simple inductive Turing machines.

Definitions imply the following results.

Proposition 3.2. *For any inductive Turing machine M, we have:*

a) $L^{bt}(M) \subseteq L^{ot}(M)$.

b) $L^{bt}(M) \subseteq L^{st}(M)$.

c) $L_{bt}(M) \subseteq L_{ot}(M)$.

d) $L_{bt}(M) \subseteq L_{st}(M)$.

Corollary 3.1. *The following inclusions are true:*

a) $\mathbf{L}^{bt}(\text{ITM1}) \subseteq \mathbf{L}^{ot}(\text{ITM1})$.

b) $\mathbf{L}_{bt}(\text{ITM1}) \subseteq \mathbf{L}_{ot}(\text{ITM1})$.

c) $\mathbf{L}_{bt}(\text{ITM1}) \subseteq \mathbf{L}_{st}(\text{ITM1})$.

d) $\mathbf{L}^{bt}(\text{ITM1}) \subseteq \mathbf{L}^{st}(\text{ITM1})$.

e) $\mathbf{L}^{bt}(\text{SITM}) \subseteq \mathbf{L}^{ot}(\text{SITM})$.

f) $\mathbf{L}_{bt}(\text{SITM}) \subseteq \mathbf{L}_{ot}(\text{SITM})$.

g) $\mathbf{L}_{bt}(\text{SITM}) \subseteq \mathbf{L}_{st}(\text{SITM})$.

h) $\mathbf{L}^{bt}(\text{SITM}) \subseteq \mathbf{L}^{st}(\text{SITM})$.

Computation by output stabilizing is the basic mode of inductive Turing machines when they perform computations (Burgin, 2005*a*). Properties of inductive Turing machines show that an inductive Turing machine M of the first order that accepts (or computes) by halting is linguistically equivalent to an accepting (or computing) Turing machine (Burgin, 2005*a*). At the same time, in general inductive Turing machines of the first order are essentially more powerful than Turing machines. For instance, there are simple inductive Turing machines that solve the halting problem for all Turing machines. This gives us the following result.

Proposition 3.3. *For any inductive Turing machine M, we have:*

a) Computation by state stabilizing is not linguistically equivalent to computation by halting in the class of all inductive Turing machines of the first order.

b) Acceptation by state stabilizing is not linguistically equivalent to acceptation by halting in the class of all inductive Turing machines of the first order.

Note that halting is a very specific case of stabilizing in which the process simply stops. However, it is more natural to compare non-stopping processes of acceptation and computation for inductive Turing machines.

Comparing the state stabilizing computed language $L^{st}(M)$ of an inductive Turing machine M and the output stabilizing computed language $L^{ot}(M)$ of the machine M, we see that in general these languages do not coincide. The same can be true for the accepted languages of the inductive Turing machine M. Such machines are considered in the following examples.

Example 3.1. Let us take a simple inductive Turing machine M such that works in the alphabet $\{0, 1\}$ and given a word w as its input, changes w to the word $w1$, i.e., M writes 1 at the end of w, gives as the output and repeats this operation with the output without stopping. As the output of M is changing on each step, the output stabilizing accepted language $L_{ot}(M)$ of the machine M is empty. At the same time, if we take all states of M as a single final group, then the state stabilizing accepted language $L_{st}(M)$ of the machine M contains all words in the alphabet $\{0, 1\}$. Thus, $Lot(M) \neq Lst(M)$.

The same inequality is true for the computed language of the inductive Turing machine M. Indeed, the output stabilizing computed language $L^{ot}(M)$ of the machine M is empty because its output never stops changing. At the same time, the machine M computes by the state stabilizing all words in the alphabet $\{0, 1\}$ that has 1 at the end. Thus, $L^{ot}(M) \neq L^{st}(M)$.

In Example 3.1, as we can see, the state stabilizing accepted language $L_{st}(M)$ of the machine M is much larger than the output stabilizing accepted language $L_{ot}(M)$ of the same machine M. The same inequality is true for the computed languages of inductive Turing machine M. However, the opposite situation is also possible.

Example 3.2. Let us take a simple inductive Turing machine W such that works in the alphabet $\{0, 1\}$ and given a word w as its input, gives this word w as the output and repeats this operation with the output and then never produces any new output. At the same time, it is possible to program the machine W so that after it rewrites w into its output tape, the machine W goes into an infinite cycle changing the state on each step. As the result, the output stabilizing accepted language $L_{ot}(W)$ of the machine W contains all words in the alphabet $\{0, 1\}$. However, if we do not define any final group in the states of the machine W, then the state stabilizing accepted language $L_{st}(W)$ of the machine W is empty. Thus, $L_{ot}(W) \neq L_{st}(W)$.

The same inequality is true for the computed languages of inductive Turing machine W. Indeed, the state stabilizing computed language $L^{ot}(W)$ of the machine W is empty because its state never stops changing and there are no final state groups. At the same time, the machine W computes by the output stabilizing all words in the alphabet $\{0, 1\}$ that has 1 at the end.Thus, $L^{ot}(W) \neq L^{st}(W)$.

Thus, the output stabilizing accepted language $L_{ot}(W)$ of the machine W is much larger than the state stabilizing accepted language $L_{st}(W)$ of the same machine W. The same inequality is true for the computed languages of inductive Turing machine W.

However, for the classes of the output stabilizing accepted by inductive Turing machines of the first order languages and the state stabilizing accepted by inductive Turing machines of the first order languages this is not true.

Theorem 3.1. *If a language L has an inductive Turing machine of the first order that computes L by output stabilizing, then L has an inductive Turing machine of the first order that computes L by bistabilizing.*

Proof. Let us consider an inductive Turing machine M of the first order that computes a language L by output stabilizing and construct an inductive Turing machine of the first order that computes a language L by bistabilizing. In the theory of inductive Turing machines, it is proved that any an inductive Turing machine of the first order is equivalent to a simple inductive Turing machine M that never stops given some input (Burgin, 2005*a*). Thus, it is possible to assume that M is a simple inductive Turing machine that never stops given some input and accepts by output stabilizing. Note that in the mode of output stabilizing final groups of states do not play any role.

Thus, it is possible to use the same machine M for bistabilizing computation of L. Indeed, making the set Q of all states of M as one final group, we see that according to Definition 11, the machine M computes the same language $L(M)$ as before by bistabilizing because the state is always stabilized. Theorem is proved. □

Results from (Burgin, 2005*a*) show that $\mathbf{L}_{st}$(ITM1) = $\mathbf{L}_{st}$(SITM) and $\mathbf{L}_{ot}$(ITM1)= $\mathbf{L}_{ot}$(SITM). Thus, Theorem 3.1 implies the following result.

Corollary 3.2. $\mathbf{L}^{ot}$(ITM1) $\subseteq$ $\mathbf{L}^{bt}$(ITM1).

This result shows that computation by bistabilizing looks more powerful than computation by output stabilizing in the class of all inductive Turing machines of the first order. However, the inverse of Theorem 3.1 is also true.

Theorem 3.2. *If a language L has an inductive Turing machine M of the first order that computes L by bistabilizing, then L has an inductive Turing machine K of the first order that computes L by output stabilizing.*

Proof. Let us consider an inductive Turing machine M of the first order that computes a language L by bistabilizing and construct an inductive Turing machine K of the first order that computes a language L by output stabilizing. As before, it is possible to assume that M is a simple inductive Turing machine.

To allow functioning in the bistabilizing mode, in the set Q of states of M, several subsets $F_1, \ldots, F_k$ are selected as final groups of states of the inductive Turing machine M.

By Proposition 3.2, $L = L^{bt}(M) \subseteq L^{ot}(M)$, i.e., the language $L^{ot}(M)$ computable by output stabilizing may have words that do not belong to the language $L^{bt}(M)$ computable by bistabilizing. To prove the necessary result, we show how to get rid of these extra words by appropriately changing the machine M.

Such an extra word w is computed by M when the output stabilizes but the state does not remain in the same final group. To prevent this, we add new symbols $b_1, b_2, b_3, \ldots, b_m$ to the alphabet $A = \{a_1, a_2, a_3, \ldots, a_m\}$ of the machine M. Then we consider all instructions of the form

$$q_h(a_{i2}, a_{i3}) \to (a_{j2}, a_{i3})q_k(T_2, T_3)$$

where q_h belongs to some final group of states Ft , while q_k does not belong to this group and change this instruction to the two following instructions

$$q_h(a_{i2}, a_{i3}) \to (a_{j2}, b_{i3})q_k(T_2, T_3)$$
$$q_k(a_{i2}, b_{i3}) \to (a_{j2}, a_{i3})q_k(T_2, T_3)$$

We do not change other instructions of M and in such a way, we obtain a simple inductive Turing machine K. As a result of these changes, the output of K always changes when the sate leaves some final group. So, the output stabilizes if and only if the state stabilizes in some final group. Consequently, the new inductive Turing machine K computes the given language L. Theorem is proved. □

Theorem 3.2 implies the following result.

Corollary 3.3. $\mathbf{L}^{bt}$(ITM1) $\subseteq$ $\mathbf{L}^{ot}$(ITM1).

This result shows that computation by output stabilizing looks more powerful than computation by bistabilizing in the class of all inductive Turing machines of the first order. However, Theorems 3.1 and 3.2 together imply that these modes are linguistically equivalent.

Theorem 3.3. *Computation by output stabilizing is linguistically equivalent to computation by bistabilizing in the class of all inductive Turing machines of the first order.*

Corollary 3.4. $\mathbf{L}^{bt}(\text{ITM1}) = \mathbf{L}^{ot}(\text{ITM1})$.

Let us consider relations between computed and accepted languages.

Theorem 3.4. *If a language L has an inductive Turing machine of the first order that computes L by state stabilizing, then L has an inductive Turing machine of the first order that computes L by bistabilizing.*

Proof. Let us consider an inductive Turing machine M of the first order that computes a language L by state stabilizing and construct an inductive Turing machine K of the first order that computes a language L by bistabilizing. As before, it is possible to assume that M is a simple inductive Turing machine.

To allow functioning in the state stabilizing mode, in the set Q of states of M, several subsets $F_1, \ldots, F_k$ are selected as final groups of states of the inductive Turing machine M.

We begin our construction of the machine K by adding one more working tape to the tapes the machine M. A simple inductive Turing machine has only three tapes (see Section 2). However, by the standard technique described, for example, in (Hopcroft *et al.*, 2001) or in (Sipser, 1996), it is possible to show that we can build a simple inductive Turing machine which simulates two working tapes using only one working tape and accepting the same language. Thus, adding one more working tape, we do not extend the class of accepted languages.

As a result of this action, the machine K has an input tape T_{in} , an output tape T_{out} and two working tapes T_{1w} and T_{2w} . We use the tape T_{1w} for exact modeling of the working tape T_w of the machine M. To do this, we preserve all parts of the rules that are related to the tape T_w in the machine M.

At the same time, we use the tape T_{2w} for exact modeling of the output tape T_{Mout} of the machine M. To do this, we redirect all parts of the rules that are related to the tape T_{Mout} in the machine M, making them the rules for the tape T_{2w} in K.

Besides, we add the rewriting state r to the set of states of the machine M and new rules for the tape T_{out} in K. The rules in which change of the state goes in one and the same final group are preserved in K. This allows the following operations. When the state of M leaves a final group, the same happens with the machine K according to its rules. While the machine M continues its functioning, comes to the rewriting state r, erases everything from the tape Tout in K and then rewrites the word from the tape T_{2w} into the output tape T_{out} . When the state of M is outside any final group and changes, the machine K repeats the same steps.

Thus, the output of K always changes when the state leaves some final group or is outside any final group and changes. As a result, the output of K stabilizes if and only if the state remains inside some final group. Besides, it stabilizes on the first word that was on the output tape of M when the stabilization of states started. So, the machine K computes the language L by bistabilizing. Theorem is proved. □

Theorem 3.4 implies the following result.

Corollary 3.5. $\mathbf{L}^{st}(\mathrm{ITM1}) \subseteq \mathbf{L}^{bt}(\mathrm{ITM1})$.

Note that the machine K constructed in the proof of Theorem 3.4 also computes the language L by output stabilizing. This gives us the following result.

Theorem 3.5. *If a language L has an inductive Turing machine of the first order that computes L by state stabilizing, then L has an inductive Turing machine of the first order that computes L by output stabilizing.*

Theorem 3.5 implies the following result.

Corollary 3.6. $\mathbf{L}^{st}(\mathrm{ITM1}) \subseteq \mathbf{L}^{ot}(\mathrm{ITM1})$.

Inversion of Theorem 3.4 is also true.

Theorem 3.6. *If a language L has an inductive Turing machine of the first order that computes L by bistabilizing, then L has an inductive Turing machine of the first order that computes L by state stabilizing.*

Proof. Let us consider an inductive Turing machine M of the first order that computes a language L by bistabilizing and construct an inductive Turing machine K of the first order that computes a language L by state stabilizing. As before, it is possible to assume that M is a simple inductive Turing machine.

To allow functioning in the bistabilizing mode, in the set $Q = \{q_1, q_2, q_3, \ldots, q_m\}$ of states of M, several subsets $F_1, \ldots, F_k$ are selected as final groups of states of the inductive Turing machine M.

By Proposition 3.2, $L = L^{bt}(M) \subseteq L^{st}(M)$, i.e., the language $L^{st}(M)$ computable by output stabilizing may have words that do not belong to the language $L^{bt}(M)$ computable by bistabilizing. To prove the necessary result, we show how to get rid of these extra words by appropriately changing the machine M.

Such an extra word w is computed by M when the state stabilizes in some final group but the output does not remain the same all the time. To prevent this, we add a new states $p_1, p_2, p_3, \ldots, p_m$ to the set Q of states of the machine M without changing the final groups. Then we consider all instructions of the form.

$$q_h(a_{i2}, a_{i3}) \to (a_{j2}, a_{j3})q_k(T_2, T_3)$$

where q_h and q_k belong to some final group of states F_n but $a_{i3} \neq a_{j3}$ and change this instruction to the two following instructions

$$q_h(a_{i2}, a_{i3}) \to (a_{j2}, a_{j3})p_k(T_2, T_3)$$

$$p_k(a_{i2}, a_{j3}) \to (a_{j2}, a_{i3})q_k(T_2, T_3)$$

We do not change other instructions of M and in such a way, we obtain a simple inductive Turing machine K. As a result of these changes, if the sate belongs to a final group, it always leaves this group when the output of K changes. So, the state stabilizes in some final group if and only if the output stabilizes. Consequently, the new inductive Turing machine K computes the given language L by state stabilizing. Theorem is proved. □

Theorem 3.6 implies the following result.

Corollary 3.7. $\mathbf{L}^{bt}(\text{ITM1}) \subseteq \mathbf{L}^{st}(\text{ITM1})$.

This result shows that computation by output stabilizing looks more powerful than computation by bistabilizing in the class of all inductive Turing machines of the first order. However, Theorems 3.4 and 3.6 together imply that these modes are linguistically equivalent.

Theorem 3.7. *Computation by state stabilizing is linguistically equivalent to computation by bistabilizing in the class of all inductive Turing machines of the first order.*

Corollary 3.8. $\mathbf{L}^{bt}(\text{ITM1}) = \mathbf{L}^{st}(\text{ITM1})$.

Theorems 3.3 and 3.7 together imply that all considered computing modes are linguistically equivalent.

Theorem 3.8. *Computation by state stabilizing, computation by output stabilizing and computation by bistabilizing are linguistically equivalent in the class of all inductive Turing machines of the first order.*

Corollary 3.9. $\mathbf{L}^{bt}(\text{ITM1}) = \mathbf{L}^{st}(\text{ITM1}) = \mathbf{L}^{ot}(\text{ITM1})$.

Inductive Turing machines allow modeling Turing machines, namely, for any Turing machine T, there is an inductive Turing machine M that has the same language as P (Burgin, 2005*a*). This makes possible to obtain the classical result of computability theory that computing by final state is linguistically equivalent to computing by halting (Burgin, 2005*a*; Hopcroft *et al.*, 2001; Sipser, 1996) as a direct corollary of Theorem 3.8.

Comparing the state stabilizing computed language $L^{st}(M)$ of an inductive Turing machine M, the output stabilizing computed language $L^{ot}(M)$ of the machine M and the bistabilizing computed language $L^{ot}(M)$ of the machine M, we see that in general these languages do not coincide. The same can be true for the accepted languages of the inductive Turing machine M. Such machines are considered in the following examples.

Indeed, the output stabilizing accepted language $L_{ot}(W)$ of the machine W from Example 3.2 is much larger than the bistabilizing accepted language $L_{bt}(W)$ of the same machine W. The same inequality is true for the computed languages of inductive Turing machine W. In addition, the state stabilizing accepted language $L_{st}(M)$ of the machine M is much larger than the bistabilizing accepted language $L_{bt}(M)$ of the same machine M. The same inequality is true for the computed languages of inductive Turing machine M.

However, Proposition 3.2 shows that for any Turing machine M, the language $L_{bt}(M)$ cannot be larger than the language $L_{st}(M)$ and the language $L_{bt}(M)$ cannot be larger than the language $L_{ot}(M)$.

In addition, for the classes of the output stabilizing accepted by inductive Turing machines of the first order languages and the state stabilizing accepted by inductive Turing machines of the first order languages this is not true.

Now let us explore for inductive Turing machines of the first order, relations between computed and accepted languages and classes of languages.

Theorem 3.9. *If a language L has an inductive Turing machine M of the first order that accepts L by state stabilizing, then L has an inductive Turing machine V of the first order that computes L by state stabilizing.*

Proof. Let us consider an inductive Turing machine M of the first order that accepts a language L by state stabilizing, i.e., $L = L_{st}(M)$. As it is proved that any an inductive Turing machine of the first order is equivalent to a simple inductive Turing machine M (Burgin, 2005*a*), it is possible to assume that M is a simple inductive Turing machine.

Then we change the rules R of the inductive Turing machine M to the rules P of the inductive Turing machine V by the following transformation. We exclude from all rules of the machine M any possibility to write something into the output tape. Besides, we add such rules according to which the process of functioning machine V begins so that the output head writes the input word into the output tape.

By construction, V is also a simple inductive Turing machine.

As there no other transformations of the system of initial rules, the output of V is never changing. By Definition 3.3, this output is the result of computation for V if and only if after some number of steps, the state of the machine M always remains in the same final group of states. It means that a word w is accepted by state stabilizing in the inductive Turing machine M if and only if the word w is computed by state stabilizing in the inductive Turing machine V. Consequently, languages $L_{st}(M)$ and $L^{st}(V)$ coincide. Theorem is proved. □

Corollary 3.10. $\mathbf{L}_{st}(\mathrm{ITM1}) \subseteq \mathbf{L}^{st}(\mathrm{ITM1})$.

This result shows that computation by state stabilizing looks more powerful than acceptation by state stabilizing in the class of all inductive Turing machines of the first order.

Theorem 3.10. *If a language L has an inductive Turing machine M of the first order that computes L by output stabilizing, then there is an inductive Turing machine W of the first order that accepts L by output stabilizing.*

Proof. Let us consider an inductive Turing machine M of the first order that computes a language L by output stabilizing, i.e., $L = L^{ot}(M)$. As it is proved that any an inductive Turing machine of the first order is equivalent to a simple inductive Turing machine M (Burgin, 2005*a*), it is possible to assume that M is a simple inductive Turing machine.

In addition, we consider a Turing machine G that given a word 1^n, generates n different words in the alphabet X of the machine M, generating all words in the alphabet X in such a way. We also take a Turing machine C that compares its input with the word written in the tape of this machine and called the sample word of C. When both words are equal, the machine C gives 1 as its output and halts. Otherwise, the machine C gives 0 as its output and halts.

This allows us to build the machine W in the following way. It contains subroutines G_0 , C_0 and M_0 , that simulate the machines G, C and M, respectively. The machine W has as many working tapes as it is necessary for functioning of the subroutines G_0 , C_0 and M_0. In particular, two counting tapes are added - the counting tape for the machine W and the counting tape for the subroutine M_0. In these tapes, numbers of iterations are stored. Then we add rules for W such that allow it to perform the following steps.

1. When a word w comes to W as its input, the machine W writes w into the working tape of the machine C as its sample word and goes to the step 2.

2. The machine W writes the number 1 into the counter tape, which contains the number of iterations, and goes to the step 3.

3. The machine W gives the number 1 as the input to the subroutine G_0 and goes to the step 4.

4. The subroutine G_0 generates the word u_1 and gives it as the input to the subroutine M_0, going to the step 5.

5. The subroutine M_0 computes with this input until the first word w_1 appears in the output tape of M_0. Then the machine W goes to the step 6.

6. The machine W writes the word w_1 in its output tape and gives w_1 as the input to the subroutine C_0, which compares w_1 and w. Then the machine W goes to the step 7 or 9 depending on the output of C_0.

7. If the output of C_0 is 1, then the subroutine M_0 continues its computations until the next output word, in this case w_2, appears in the output tape of M_0. Then the machine W goes to the step 8.

8. The machine W gives w_2 as the input to the subroutine C_0, which compares w_2 and w. Then the machine W goes to the step 7 or 9 depending on the output of C_0. Note that if the output of C_0 is always 1 starting from some input, the machine W accepts w by output stabilizing.

9. If the output of C_0 is 0, then the machine W writes the word w in its output tape, changes this word to the word checked by C_0 (it will be w_1 if the previous step had number 6, while it will be w_2 if the previous step had number 8) and goes to the step 10.

10. The machine W adds 1 to the number in its counter tape, gives this new number n (in the second iteration $n = 2$) as the input to the subroutine G_0 and goes to the step 11.

11. The subroutine G_0 generates n words $u_1, u_2, \ldots, u_n$ and gives all of them one by one as the inputs to the subroutine M_0.

12. The subroutine M_0 writes 1 into its counting tape and computes with the input u_1 until it writes n words into the output tape. Then the machine W goes to the step 13.

13. The machine W gives the current output word w_k of the subroutine M_0 as the input to the subroutine C_0 , which compares w_k and w. Then the machine W goes to the step 14 or 15 depending on the output of C_0.

14. If the output of C_0 is 1, then the subroutine M_0 continues its computations until the next output word, say w_r, appears in the output tape of M_0. Note that if the output of C_0 is always 1 starting from some input, the machine W accepts w by output stabilizing.

15. If the output of C_0 is 0 and the number t in its counting tape is less than k, then the machine W writes the word w in its output tape and changes this word to the word checked by C_0, while the subroutine M_0 adds 1 to the number t in its counting tape and computes with the input u_{t+1} until it writes n words into the output tape. Then the machine W goes to the step 13.

16. If the number t in its counting tape of M_0 becomes equal to k, the machine W goes to the step 10.

If the word w is computed by output stabilizing of the inductive Turing machine M, then given some input u, the machine M makes some number of steps, and then its output becomes equal to w and stops changing. In this case, the output of C_0 is always 1 starting after some number of steps, and consequently, the machine W accepts w by output stabilizing.

If the word w is not computed by output stabilizing of the inductive Turing machine M for any input, then either the output of M is not stabilizing or it is stabilizing with the word v that is not equal to w. In the first case, the output of W is also not stabilizing and thus, the machine W does not accept w by output stabilizing. In the second case, the output of W is also not stabilizing because both words w and v appear infinitely many times in the output tape of W and thus, the machine W does not accept w by output stabilizing.

By construction, W is a simple inductive Turing machine.

Thus, the simple inductive Turing machine W does not accept w by output stabilizing is and only if the simple inductive Turing machine M does not accept w by output stabilizing.

Theorem is proved because w is an arbitrary word in the alphabet of the inductive Turing machine M. □

Corollary 3.11. $\mathbf{L}^{ot}(\mathrm{ITM1}) \subseteq \mathbf{L}_{ot}(\mathrm{ITM1})$.

This result shows that acceptation by output stabilizing looks more powerful than computation by output stabilizing in the class of all inductive Turing machines of the first order.

Let us take an inductive Turing machine M functioning of which satisfies Condition ST and its state stabilizing language $L^{st}(M)$, i.e., the set of all words computed by state stabilizing of the machine M (cf. Definition 3.3). Thus, a word w is computed by state stabilizing of the machine M if and only if is computed by output stabilizing of the machine M. Consequently, $L^{st}(M) = L^{ot}(M)$. This gives us the following results.

Theorem 3.11. *For any inductive Turing machine M of the first order functioning of which satisfies Condition ST,* $\mathrm{L}^{st}(\mathrm{M}) \subseteq \mathrm{L}^{ot}(\mathrm{M})$.

Corollary 3.12. $\mathbf{L}^{st}(\mathrm{ITM1}) \subseteq \mathbf{L}^{ot}(\mathrm{ITM1})$.

This result shows that computation by output stabilizing looks more powerful than computation by state stabilizing in the class of all inductive Turing machines of the first order.

Note that in general Theorem 11 does not imply equality of the classes $\mathbf{L}^{st}(\mathrm{ITM1})$ and $\mathbf{L}^{ot}(\mathrm{ITM1})$ because not all inductive Turing machines satisfy Condition ST.

At the same time, other results obtained in this paper allows us to find more exact relations between these classes of languages. Namely, by Corollary 3.1, we have $\mathbf{L}_{st}(\mathrm{ITM1}) \subseteq \mathbf{L}_{ot}(\mathrm{ITM1})$.

By Corollary 3.2, we have $\mathbf{L}^{ot}(\mathrm{ITM1}) \subseteq \mathbf{L}^{bt}(\mathrm{ITM1})$.

Comparing acceptation by state with acceptation by output, we obtain the following inclusion $\mathbf{L}_{ot}(\mathrm{ITM1}) \subseteq \mathbf{L}_{st}(\mathrm{ITM1})$.

By Corollary 3.4, we have $\mathbf{L}^{bt}(\mathrm{ITM1}) \subseteq \mathbf{L}^{ot}(\mathrm{ITM1})$.

By Corollary 3.10, we have $\mathbf{L}_{st}(\mathrm{ITM1}) \subseteq \mathbf{L}^{st}(\mathrm{ITM1})$.

By Corollary 3.11, we have $\mathbf{L}^{ot}(\mathrm{ITM1}) \subseteq \mathbf{L}_{ot}(\mathrm{ITM1})$.

By Corollary 3.12, we have $\mathbf{L}^{st}(\mathrm{ITM1}) \subseteq \mathbf{L}^{ot}(\mathrm{ITM1})$.

This gives us the following chain of inclusions:

$\mathbf{L}^{st}(\mathrm{ITM1}) \subseteq \mathbf{L}^{ot}(\mathrm{ITM1}) \subseteq \mathbf{L}_{ot}(\mathrm{ITM1}) \subseteq \mathbf{L}_{st}(\mathrm{ITM1}) \subseteq \mathbf{L}^{st}(\mathrm{ITM1})$.

By properties of sets and the inclusion relation, the chain (5) implies that all inclusions in it are equalities. Thus, we have the following result.

Theorem 3.12. *In the class of all inductive Turing machines of the first order, the following modes of functioning are linguistically equivalent:*

1. *Acceptation by state stabilizing.*
2. *Acceptation by output stabilizing.*
3. *Computation by state stabilizing.*
4. *Computation by output stabilizing.*

Inductive Turing machines allow modeling Turing machines, namely, for any Turing machine T, there is an inductive Turing machine M that has the same language as P (Burgin, 2005*a*). This makes possible to obtain the following classical result of computability theory as a direct corollary of Theorem 3.12.

Proposition 3.4. . *For Turing machines, the acceptation mode is linguistically equivalent to the computation mode, i.e., the class of languages accepted by Turing machines coincides with the class of languages computed by Turing machines.*

This result justifies the situation when in the majority of textbooks on computer science, it is assumed that Turing machines work in the acceptation mode and model computers and this allows modeling computers although computers, as a rule, work in the computation mode.

4. Conclusion

Various models of computation are studied in computer science - deterministic and nondeterministic finite automata, deterministic and nondeterministic pushdown automata, deterministic and nondeterministic Turing machines with one or many tapes, which can by one-dimensional and many-dimensional, and so on.

The basic results in this area are theorems on linguistic equivalence of different models of computation, i.e., equivalence with respect to the languages that are accepted/generated by these models. Thus, for finite automata, it is proved that the class of languages accepted by deterministic finite automata is the same as the class of languages accepted by nondeterministic finite automata

and as the class of languages accepted by nondeterministic finite automata with ε-transitions (cf., for example, (Hopcroft *et al.*, 2001): Theorem 2.12, Theorem 2.22; (Sipser, 1996): Theorem 1.19)).

In the theory of pushdown automata, it is proved that the class of languages accepted by pushdown automata by final state is the same as the class of languages accepted by pushdown automata by empty stack and as the class of languages generated by context free grammars (cf., for example, (Hopcroft *et al.*, 2001): Theorem 6.9, Theorem 6.11, Theorem 6.14; (Sipser, 1996): Theorem 2.12).

In the theory of Turing machines, it is proved that the class of languages accepted by deterministic Turing machines with a single tape is the same as the class of languages accepted by nondeterministic Turing machines with a single and as the class of languages accepted by Turing machines with many tapes and as the class of languages accepted by Turing machines with multidimensional tapes and as the class of languages accepted by pushdown automata with two and more stacks (cf., for example, (Hopcroft *et al.*, 2001): Theorem 8.9, Theorem 8.11, Theorem 6.14, Theorem 8.13; (Sipser, 1996): Theorem 3.8, Theorem 3.10).

In this paper, we obtained similar results for inductive Turing machines. Namely, it is proved that: (1) the class of languages computed by output stabilizing of inductive Turing machines of the first order is the same as the class of languages computed by bistabilizing of inductive Turing machines of the first order (Theorem 3.3); (2) the class of languages computed by output stabilizing of inductive Turing machines of the first order is the same as the class of languages computed by state stabilizing of inductive Turing machines of the first order (Theorem 3.8); (2) the class of languages computed by output (state) stabilizing of inductive Turing machines of the first order is the same as the class of languages accepted by output (state) stabilizing of inductive Turing machines of the first order (Theorem 3.12).

It is necessary to remark that it is possible to include the results of this paper into a standard course of the theory of automata, formal languages and computation.

The obtained results bring us to the following problems.

Problem 1. Study different modes of functioning for inductive Turing machines of the higher orders.

Problem 2. Study inductive Turing machines in which the control device is a more powerful automaton than a finite automaton.

Here we considered accepting and computing modes of inductive Turing machine brings us to the following problem.

Problem 3. For inductive Turing machines, study relations between the decision mode, accepting mode and computing mode of functioning.

It would be important to study properties of stabilizing computations in distributed systems. Grid automata provide the most advanced and general model of distributed systems (Burgin, 2005*a*).

Problem 4. Study properties of stabilizing computations for grid automata.

There are models of computation without explicit utilization of automata (cf., for example, (Milner, 1989; Lee & Sangiovanni-Vincentelli, 1996; Burgin & Smith, 2010).

Problem 5. Study properties of stabilizing computations utilizing models of concurrent computational processes.

References

Beros, A. A. (2013). Learning theory in the arithmetical hierarchy, preprint in mathematics. *math.LO/1302.7069 (electronic edition: http://arXiv.org)*.

Büchi, J. Richard (1960). Weak second-order arithmetic and finite automata. *Z. Math. Logik und Grundl. Math.* **6**, 66–92.

Burgin, M. (1999). Super-recursive algorithms as a tool for high performance computing. *Proceedings of the High Performance Computing Symposium, San Diego* **6**, 224–228.

Burgin, M. (2003). Nonlinear phenomena in spaces of algorithms. *International Journal of Computer Mathematics* **80**(12), 1449–1476.

Burgin, M. (2004). Algorithmic complexity of recursive and inductive algorithms. *Theoretical Computer Science* **317**(13), 31 – 60.

Burgin, M. (2005*a*). *Super-recursive Algorithms*. Springer, New York.

Burgin, M. (2005*b*). Superrecursive hierarchies of algorithmic problems. In: *Proceedings of the 2005 International Conference on Foundations of Computer Science, CSREA Press, Las Vegas*. pp. 31–37.

Burgin, M. (2006). Algorithmic control in concurrent computations. *Proceedings of the 2006 International Conference on Foundations of Computer Science, CSREA Press, Las Vegas* **6**, 17–23.

Burgin, M. (2007). Algorithmic complexity as a criterion of unsolvability. *Theoretical Computer Science* **383**(2/3), 244 – 259.

Burgin, M. (2010*a*). Algorithmic complexity of computational problems. *International Journal of Computing & Information Technology.* **2**(1), 149–187.

Burgin, M. (2010*b*). *Measuring power of algorithms, computer programs, and information automata*. Nova Science Publishers, New York.

Burgin, M. and A. Klinger (2004). Experience, generations, and limits in machine learning. *Theoretical Computer Science* **317**(1/3), 71 – 91.

Burgin, M. and B. Gupta (2012). Second-level algorithms, superrecursivity, and recovery problem in distributed systems. *Theory of Computing Systems* **50**(4), 694 – 705.

Burgin, M. and E. Eberbach (2008). Cooperative combinatorial optimization: Evolutionary computation case study. *Biosystems* **91**(1), 34 – 50.

Burgin, M. and E. Eberbach (2009*a*). On foundations of evolutionary computation: An evolutionary automata approach. *Handbook of Research on Artificial Immune Systems and Natural Computing: Applying Complex Adaptive Technologies, Hongwei Mo, Ed., IGI Global, Hershey, Pennsylvania,* pp. 342–360.

Burgin, M. and E. Eberbach (2009*b*). Universality for Turing machines, inductive Turing machines and evolutionary algorithms. *Fundamenta Informaticae* **91**, 53–77.

Burgin, M. and E. Eberbach (2010). Bounded and periodic evolutionary machines. *Proc. 2010 Congress on Evolutionary Computation (CEC'2010), Barcelona, Spain* pp. 1379–1386.

Burgin, M. and E. Eberbach (2012). Evolutionary automata: Expressiveness and convergence of evolutionary computation. *The Computer Journal* **55**(9), 1023–1029.

Burgin, M. and M. L. Smith (2010). A theoretical model for grid, cluster and internet computing. *Selected Topics in Communication Networks and Distributed Systems, World Scientific, New York/London/Singapore* pp. 485 – 535.

Burgin, M. and N. Debnath (2004). Measuring software maintenance. *Proceedings of the ISCA 19th International Conference "Computers and their Applications", ISCA, Seattle, Washington* pp. 118–121.

Burgin, M. and N. Debnath (2005). Complexity measures for software engineering. *J. Comp. Methods in Sci. and Eng.* **5**(1 Supplement), 127–143.

Burgin, M. and N. Debnath (2009). Super-recursive algorithms in testing distributed systems. *Proceedings of the ISCA 24-th International Conference "Computers and their Applications", ISCA, New Orleans, Louisiana, USA* pp. 209–214.

Burgin, M., N. Debnath and H. K. Lee (2009). Measuring testing as a distributed component of the software life cycle. *Journal of Computational Methods in Science and Engineering* **9**(Supplement 2/ 2009), 211–223.

Burks, A. W. and J. B. Wright (1953). Theory of logical nets. *Proceedings of the IRE* **41**(10), 1357–1365.

Calinescu, R., R. France and C. Ghezzi (2013). Editorial. *Computing* **95**(3), 165–166.

Chadha, R., A. P. Sistla and M. Viswanathan (2009). Power of randomization in automata on infinite strings. In: *CONCUR 2009 - Concurrency Theory* (Mario Bravetti and Gianluigi Zavattaro, Eds.). Vol. 5710 of *Lecture Notes in Computer Science*. pp. 229–243. Springer, Berlin/Heidelberg.

Gold, E. M. (1967). Language identification in the limit. *Information and Control* **10**(5), 447 – 474.

Hopcroft, J. E., R. Motwani and J. D. Ullman (2001). *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, Boston/San Francisco/New York.

Lee, E.A. and A. Sangiovanni-Vincentelli (1996). Comparing models of computation. In: *Computer-Aided Design, 1996. ICCAD-96. Digest of Technical Papers., 1996 IEEE/ACM International Conference on*. pp. 234–241.

Milner, R. (1989). *Communication and Concurrency*. Prentice-Hall, Inc.. Upper Saddle River, NJ, USA.

Rogers, Jr., Hartley (1987). *Theory of Recursive Functions and Effective Computability*. MIT Press. Cambridge, MA, USA.

Sipser, M. (1996). *Introduction to the Theory of Computation*. 1st ed.. International Thomson Publishing.

Thomas, Wolfgang (1990). Handbook of theoretical computer science (vol. b). Chap. Automata on Infinite Objects, pp. 133–191. MIT Press. Cambridge, MA, USA.