In "Artificial Intelligence and Computer Science", Nova Science Publishers, New York, 2005, pp. 1-61

Measuring Power of Algorithms, Programs, and Automata

Mark Burgin
Department of Mathematics
University of California, Los Angeles
405 Hilgard Ave.
Los Angeles, CA 90095

Abstract

We are living in a world where complexity of systems created and studied by people grows beyond all imaginable limits. Computers, their software and their networks are among the most complicated systems of our time. Science is the only efficient tool for dealing with this overwhelming complexity. One of the methodologies developed in science is the axiomatic approach. It proved to be very powerful in mathematics. In this paper, we develop further an axiomatic approach in computer science initiated by Manna, Blum and other researchers. In the traditional constructive setting, different classes of algorithms (programs, processes or automata) are studied separately, with some indication of relations between these classes. Thus, the constructive approach gave birth to the theory of Turing machines, theory of partial recursive functions, theory of finite automata, and other theories of constructive models of algorithms. The axiomatic context allows one to research classes of classes of algorithms, automata, and processes. As a result, axiomatic approach goes higher in the hierarchy of computer and network models, reducing in such a way complexity of their study. The suggested axiomatic methodology is applied to evaluation of possibilities of computers and their networks. People more and more rely on computers and other information processing systems. So, it is vital to know better than now what computers and other information processing systems can do and what they can't do. The main emphasis is done on such properties as computability, decidability, and acceptability.

Key words: computation, computing power, accepting power, axiom, solvability, computability, decidability, acceptability

1. Introduction

Mathematical methods play more and more important role in society. Mathematics is applied to a diversity of other fields. Mathematics provides a variety of methods for description, modeling, computation, reasoning, constructing etc. This extensive variety of methods is traditionally divided into two directions: constructive and axiomatic. Beginning from Euclid's "*Elements*," which present geometry as an axiomatic discipline, axiomatic methods have demonstrated their power in mathematics. As Burton (1997) writes, generation after generation regarded the "*Elements*" as the summit and crown of logic and mathematics, and its study as the best way of developing facility of exact reasoning. Abraham Lincoln at the age of forty, while still a struggling lawyer, mastered the first six books of Euclid, solely as training for his mind. Even now, in spite of the discovery of non-Euclidean geometries and improvements of the system of Euclid, "Elements" largely remains the supreme model of a book in mathematics, demonstrating the power of the axiomatic approach.

The main goal of this paper is to show that axiomatic methods are also very efficient for computer science. Methods and technique developed in this paper are oriented at various applications in computer science and technology. Examples of practical problems that benefit from this approach are debugging and testing computer software, design of software metrics, and comparison of computational power for different systems.

It is possible to consider three levels of axiomatization: local, global, and multiglobal.

Local or *object oriented axiomatization* gives axioms for a description/determination of a single object, e.g., a separate program or algorithm. As an example, we can take the axiomatic theory of programs suggested by Hoare (1969).

Global or *class oriented axiomatization* gives axioms for a description/determination of a definite class of objects, e.g., Euclidean geometry, vector spaces, groups.

Multiglobal or *feature oriented axiomatization* gives axioms for a description/determination of a system of definite classes that have some specific features in common, e.g., Euclidean geometry, vector spaces, groups.

In the global approach, a system of axioms is taken (built or selected) for a description of some system. Then these axioms are used for deduction of properties of this system. This is the standard mathematical way of axiomatic studies. It is embodied in such

classical works as Euclid's *Elements*, Hilbert's axiomatization of the Euclidean geometry, and axiomatic set theories (cf. Fraenkel and Bar-Hillel, 1958): **ZF** of Zermelo-Fraenkel, **VN** of von Neumann, **BG** of Bernays-Godel, and two theories of Quine – **NF** and **ML**. The problem that is solved by a global axiomatization is what are basic properties of a given system that allow one to deduce all other properties.

In contrast to this, the multiglobal approach is oriented not at a system but at definite properties. The aim of multiglobal axiomatization is to characterize by simple properties $P_1, \dots, P_m$ such classes of systems in which some important results $R_1, \dots, R_n$ are valid. Properties $P_1, \dots, P_m$ are formulated as axioms and conditions $A_1, \dots, A_m$ and the necessary results $R_1, \dots, R_n$ are deduced from these axioms and conditions $A_1, \dots, A_m$ as theorems $T_1, \dots, T_n$.

This allows one not to prove the same result, for example, R_i, for each class from a diverse variety separately, but to check for each of those classes only initial axioms and then to deduce this result R_i from a general theorem T_i proved in the axiomatic setting. Axiomatic form of concept introduction in modern mathematics, such as rings, fields, Hilbert spaces, Banach spaces etc., gives classical examples of such a technique. Another example of this approach is Blum's (1967) concept of the size of a machine, which synthesized many measures of algorithms and programs. Another his concept is computational complexity, which in the axiomatic form synthesized such concepts as time complexity and space complexity of computation.

The first Godel's incompleteness theorem (1931) shows that the problem of global axiomatization cannot be completely solved for sufficiently rich mathematical systems. Axioms cannot present all properties of such systems, given traditional means of inference. Thus, it becomes more efficient to consider partial systems of axioms that define systems of classes in the context of multiglobal axiomatization. According to the modern methodology of science such systems of axioms become laws of the second order (Burgin and Kuznetsov, 1994).

The reason why we need multiglobal axiomatization in computer science is existence of a huge diversity of computer and network systems, programs and program systems, as well as their theoretical models. Axiomatics allows one to compress information about all these devices and systems, providing efficient means for their study and development.

An extreme case of the multiglobal axiomatization is minimal mathematics aimed at finding minimal conditions for a possibility to prove some result or a system of results. Multiglobal axiomatization is also closely related to reverse mathematics, which strives to obtain the simplest conditions under which a given result (mathematical theorem) is valid. Namely, reverse mathematics is the branch of mathematics concerned with what are the minimal axioms needed to prove the particular theorem (Friedman and Hirst, 1990; Giusto and Simpson, 2000). It turns out that over a weak base theory, many mathematical statements are equivalent to the particular additional axiom needed to prove them.

Axiomatic approach brings the theory of algorithms to a new level. Historically, this theory appeared when models of algorithms were constructed to prove absolute undecidability results of some mathematical problems. Such ultimate undecidability demanded to show that there is no algorithm for problem solution. In other words, it was necessary to give an exact description of all possible algorithms. The goal was achieved by constructing such ultimate classes of recursive algorithms as Turing machines, partial recursive functions and many other models of algorithms. However, recent development of computer science demonstrated that these classes are not absolute and there are more powerful algorithms (cf., for example, Burgin, 2001). Thus, we have come to the situation when it is impossible to reduce decidability problems to one universal class of algorithms. Axiomatic setting allows one eliminate this obstacle and to prove undecidability without constructive models of algorithms.

In this paper, we consider the most conventional class of reactive algorithms and abstract automata. These algorithms and automata react to a given input by producing some output or coming to some state. Active, interactive, and proactive algorithms and automata are considered elsewhere.

The axiomatic approach brings the theory of algorithms and thus, the whole computer science to a new higher level. Historically, the theory of algorithms appeared when models of algorithms were constructed to prove undecidability of some mathematical problems. In a new setting of an axiomatic theory, proofs of undecidability do not demand constructive models of algorithms. When computers were created and utilized, the theory of algorithms formed a core of computer science and till now this is the most developed mathematical discipline of computer science.

Our advent in a new realm of multiglobal axiomatics of algorithms, automata, and programs, we begin (Section 2) with an analysis of such basic for computer science concepts as algorithms, programs, and abstract automata. The suggested approach allows us to eliminate some contradictions and inconsistencies in the conceptual system of computer science.

In Section 3, types of algorithms and functioning of automata and algorithms are analyzed and formalized. A multifaceted typology is developed for algorithms aimed at axiomatizing computer science.

In Section 4, basic axioms for algorithms are considered in three forms: postulates as the most basic assumptions, axioms as global assumptions that represent important properties, and conditions as local assumptions that represent specific properties.

An example of computer science postulates is the Deterministic Computation Postulate **PDC**, which states that any algorithm A that takes inputs from a set X and gives outputs that belong to a set Y determines a function f_A from X into Y.

An example of computer science axioms is the Universality Axiom **AU**, which states that for a class **K** of automata/algorithms and some coding **c : K ®** V^+, there is a universal algorithm/automaton in **K**.

An example of computer science conditions is the Switching Condition **SW**, which states that for any x and y from X, there is a switching for x and y algorithm/automaton in a class **K**.

In Section 5, power of algorithms and automata is investigated. Exact mathematical methods are developed for comparison and evaluation of algorithms and automata.

In Section 6, properties and related problems of algorithms and automata are classified and studied. All properties and problems are separated into several classes: linguistic, functional, description, operational, etc. Some of these classes have been intensively studied for different models of algorithms, such as finite automata, Turing machines and some others. Here these properties and problems are considered in axiomatic setting, allowing one to essentially expand the scope of applications. Other studied here properties and problems have been beyond the scope of conventional computer science although they are important for practice.

In Section 7, boundaries for algorithms and computation are found. In particular, it is proved that all non-trivial linguistic (Theorem 7.5) and functional (Theorem 7.6) properties are

undecidable for a lot of classes of algorithms. For Turing machines, this implies classical results of Rice (1951). At the same time, for other kinds of properties (e.g., operational properties), there are both decidable and undecidable non-trivial properties.

Applications of the theoretical results to software and hardware verification and testing are considered in Section 8.

Denotations and basic definitions:

N is the set of all natural numbers;

ω is the sequence of all natural numbers;

$\varnothing$ is the empty set;

The logical symbol $\forall$ means "for any";

The logical symbol $\exists$ means "there exists";

If **P** is a property, then $\neg$**P** is the complementary property, i.e., an object *x* has the property **P** if and only if it does not have the property $\neg$**P**.

R is the set of all real numbers;

A *binary relation* *T* between sets *X* and *Y* is a subset of the direct product $X \times Y$. The set *X* is called the *domain* of *T* ($X = \mathrm{D}(T)$) and *Y* is called the *codomain* of *T* ($Y = \mathrm{CD}(T)$). The *range* of the relation *T* is $\mathrm{R}(T) = \{ y ; \exists x \in X ((x, y) \in T) \}$. The *definability domain* of the relation *T* is $\mathrm{DD}(T) = \{ x ; \exists y \in Y ((x, y) \in T) \}$.

A *function* or *total function* from *X* to *Y* is a binary relation between sets *X* and *Y* in which there are no elements from *X* which are corresponded to more than one element from *Y* and to any element from *X* is corresponded some element from *Y*. Often total functions are also called everywhere defined functions.

A *partial function* *f* from *X* to *Y* is a binary relation in which there are no elements from *X* which are corresponded to more than one element from *Y*.

For a partial function *f*, its *definability domain* DD(*f*) is the set of all elements for which *f* is defined.

For any set, *S,* $\chi_S(x)$ is its characteristic function, that is, $\chi_S(x)$ is equal to 1 when $x \in S$ and is equal to 0 when $x \notin S$, and $C_S(x)$ is its partial characteristic function, that is, $C_S(x)$ is equal to 1 when $x \in S$ and is undefined when $x \notin S$.

An *alphabet* or vocabulary A of a *formal language* is a set consisting of some symbols or letters. A vocabulary is an alphabet on a higher level of hierarchy because words of a vocabulary play the same role for building sentences as symbols in an alphabet for building words. Traditionally any alphabet is a set. However, a more consistent point of view is that an alphabet is a multiset (Knuth, 1981), containing an unbounded number of identical copies of each symbol.

A *string* or *word* is a sequence of elements from the alphabet. A^* denotes the set of all finite words in the alphabet A. Usually there is no difference between strings and words. However, having a language, we speak about words of this language and not about its strings. A^{**} denotes the set of all (finite and infinite) words in the alphabet A. A^+ denotes the set of all non-empty finite words in the alphabet A. A^{++} denotes the set of all non-empty (finite and infinite) words in the alphabet A.

A *formal language* L is any subset of A^*.

The *length* $l(w)$ of a word w is the number of letters in the word w.

ε is the *empty word*.

Λ is the *empty symbol*.

When an algorithm/automaton A gives y as the result being applied to x, it is denoted by $A(x) = y$. When an algorithm/automaton A gives no result being applied to x, it is denoted by $A(x) = *$.

D(A) is the *domain* of an algorithm A, i.e., the set X of such elements that are processed by A.

DD(A) is the *definability domain* of an algorithm A, i.e., is the set X of such elements that if any of them is given as input to A, then A gives a result/output.

C(A) is the *codomain* of an algorithm A, i.e., the set Y of such elements that are tentative outputs of A.

2. Algorithms, Programs, and Abstract Automata

We begin with clarification of relations between such basic concepts of computer science as algorithms, programs, and automata.

In many cases, these terms are used interchangeably. However, computing practice and research experience show that these concepts are different. For instance, programmers and computer scientists are aware that the same algorithm can be represented in a variety of ways. Algorithms are usually represented by texts and can be expressed practically in any language, from natural languages like English or French to programming languages like C^{++}. For example, addition of binary numbers can be represented in many ways: by a Turing machine, by a formal grammar, by a program in C^{++}, in PASCAL or in FORTRAN, by a neural network, or by a finite automaton. Besides, an algorithm can be represented by software or by hardware. That is why, as it is stressed by Shore in (Buss *et al*, 2001), it is essential to understand that algorithm is different from its representation and to make a distinction between algorithms and their descriptions.

The same situation exists even for the simplest algorithms – instructions. Cleland (2001) emphasizes that "it is important to distinguish instruction-expressions from instructions." The same instruction may be expressed in many different ways, including in different languages and in different terminology in the same language. Also, some instruction may be communicated non-verbally, e.g., when one computer sends a program to another computer.

Similar situation emerges in the case of numbers and their representations. For example, the same rational number may be represented by the following fractions ½ , 2/4 , 3/6, as well as by the decimal 0.5. Number five is represented by the Arab (or more exactly, Hindu) numeral 5 in the decimal system, the sequence 101 in the binary number system, and by the symbol V in the Roman number system. There are, at least, three natural ways for separation of algorithms from their descriptions such as programs or systems of instructions.

In **the first one**, which we call the *model approach*, we chose some type **D** of descriptions (for example, Turing machines) as a model description, in which there is a one-to-one correspondence between algorithms and their descriptions. Then we introduce an equivalence relation R between different descriptions of algorithms. This relation has to satisfy two axioms:

(**DA1**) Any description of an algorithm is equivalent to some element from the model class D.

(**DA2**) Any two elements from the model class D belong to different equivalence classes.

This approach is used by Moschovakis, who considers the problem of unique representation for algorithms in his paper "What is an Algorithm?' (2001). He makes

interesting observations and persuasively demonstrates that machine models of algorithms are only models but not algorithms themselves. His main argument is that there are many models for one and the same algorithm. To remedy this, he defines algorithms as systems of mappings, building thus, a new model for algorithm. Moschovakis calls such systems of mappings, which are defined by recursive equations, *recursors*. This is an essential progress in understanding and mathematical modeling algorithms. However, this does not solve the problem of separating algorithms as something invariant from their representations. This type of representation is on higher level of abstraction than traditional representations, such as Turing machines or partial recursive functions. Nevertheless, a *recursor* (in the sense of Moschovakis) is only a model for algorithm but not an algorithm itself.

The **second way** to separate algorithms and their descriptions is called the *relational approach* and is based on an equivalence relation R between different descriptions of algorithms. Having such relation, we define algorithm as a class of equivalent descriptions. Equivalence of descriptions can be determined by some natural axioms, describing, for example, properties of operations:

> **Composition Axiom.** Composition (sequential, parallel, etc.) of descriptions represents the corresponding composition of algorithms.
>
> **Decomposition Axiom.** If a description H defines a sequential composition of algorithms A and B, a description K defines a sequential composition of algorithms C and B, and $A = C$, then H is equivalent to K.

At the same time, the equivalence relation R between descriptions can be formed on the base of computational processes. Namely, two descriptions define the same algorithm if these descriptions generate the same sets of computational processes. This definition depends on our understanding of equal processes. For example, in some cases it is natural to consider processes on different devices as different, while in other cases it might be better to treat some processes on different devices as equal.

In particular, we have the rule suggested by Cleland (2001) for instructions:

Different instruction-expressions, i.e., representations of instructions, express the same instruction only if they prescribe the same type of action.

Such structural definition of algorithm depends on organization of computational processes. For example, let us consider some Turing machine **T** and another Turing machine **Q**. The only difference between **T** and **Q** is that **Q** contains all instruction of **T** and one more

instruction that is never used in computations of the machine **Q**. Then, on one hand, it is possible to assume that this additional instruction has no influence on computational processes and thus, **T** and **Q** define one and the same algorithm. On the other hand, if a Turing machine in a course of computation always go through all instructions to choose the one to be performed, then the processes are different and consequently, **T** and **Q** define different algorithms.

The **third way** to separate algorithms and their descriptions is called the *structural approach* because a specific invariant (structure) is extracted from descriptions. This structure is called an algorithm. Here we understand structures in the sense of (Burgin, 1997). Thus, we come to the following understanding, which separates algorithm from its descriptions.

Definition 2.1. *An algorithm is a* (*finite*) *structure that contains for some performer* (*class of performers*) *exact information* (*instructions*) *that allows this performer*(*s*) *to pursue a definite goal.*

Consequently, algorithms are compressed constructive, i.e., giving enough information for realization, representations of processes. In particular, they represent intrinsic structures of computer programs. Hence, algorithm is an essence that is independent of how it happens to be represented and is similar to mathematical objects. Once the concept of algorithm is so rendered, its broader connotations virtually spell themselves out. As a result, algorithm appears as consisting of three components: structure, representation (linguistic, mechanical, electronic etc.), and interpretation.

It is important to understand that not all systems of rules represent algorithms. For example, you want to give a book to your friend Johns, who often comes to your office. So, you decide to take the book to your office (the first rule) and to give it to Johns when he comes to your office (the second rule). While these simple rules are fine for you, they are much too ambiguous for a computer. In order for a system of rules to be applicable to a computer, it must have certain characteristics. We specify these characteristics later in formal definitions of an algorithm. Now we only state that formalized functioning of complex systems (such as people) is mostly described and controlled by more general systems of rules than algorithmic structures. They are called procedures.

Definition 2.2. *A procedure is a compressed operational representation of a process.*

For example, you have a set of instructions for planting a garden where the first step instructed you to remove all large stones from the soil. This instruction may not be effective if there is a ten-ton rock buried just below ground level. So, this is not an algorithm, but only a procedure. However, if you have means to annihilate this rock, this system of rules becomes an algorithm.

It is necessary to remark that the above given definition describes procedure in the theoretical sense. There is also a notion of procedure in the sense of programming.

A *procedure in a program*, or *subroutine*, is a specifically organized sequence of instructions for performing a separate task. This allows the subroutine code to be called from multiple places of the program, even from within itself, in which case the form of computation is called recursive. Most programming languages allow programmers to define subroutines. Subroutines, or procedures in this sense, are specific representations of algorithms by means of programming la nguages.

There are three classes of representation for algorithms and procedures:

Automaton representations. Turing machines and finite automata give the most known examples of such representations.

Instruction representations. Formal grammars, rules for inference, and Post productions give the most known examples of such representations.

Equation representations. Here an example of such recursive equation is given.

$$\text{Fact}(n) = \begin{cases} 1 & \text{when} \quad n = 1, \\ n \cdot \text{Fact}(n - 1) & \text{when} \quad n > 1 \end{cases}$$

The fixed point of this recursive equation defines a program for computation of the factorial $n!$

Algorithms are connected to procedures in a general sense, being special cases of procedures. If we consider algorithms as rigid procedures, then there are also soft procedures, which have recently become very popular in the field of soft computing.

To discern algorithms from procedures, it is assumed that algorithms satisfy specific conditions:

1. *Operational decomposition*: There is a system of effective basic operations that are performed in a simple way with some basic constructive objects, while all other operations can be reduced to the basic operations.

2. *Purposefulness*: Execution of algorithms is aimed at some purpose.

3. *Discreteness*: Operations are performed in a discrete time, that is, step by step and each such step is separated from the others.

Some experts demand additional conditions that are not always satisfied both in the theory of algorithms and practice of computation:

4. *Substantial finiteness*: All objects of operation in algorithm and the number of objects involved into operation on each step are finite.
5. *Operational finiteness*: The number of operations in algorithm and operations themselves are finite.
6. *Temporal finiteness*: The result of the algorithm functioning/execution is obtained in finite time.
7. *Demonstrativeness*: Algorithm provides explicit information when it obtains the necessary result.
8. *Definability*: Given a relevant input, algorithm always obtains the result.

It is possible to formalize all these conditions in the context of a multiglobal axiomatic approach.

Like algorithms, procedures (in a general sense) are also different from their descriptions.

Definition 2.3. *A representation/description of a procedure/algorithm is a symbolic materialization of this procedure/algorithm as a structure.*

According to this understanding, algorithms and procedures are similar to mathematical objects because, as it is demonstrated in Burgin (1998), all mathematical objects are structures. This explains why algorithms in a strict sense appeared in mathematics, were named after a mathematician, and have been developed into a powerful and advanced mathematical theory – the theory of algorithms and computation.

However, in the theory of algorithms and computation there is no distinction between algorithms and their descriptions. In what follows, we follow this tradition to make comprehension easier.

Going from more abstract and implicit, it is natural to consider three levels of algorithm representations:

1. *An analytical description*.
2. *A prescriptive description*.
3. *An embodied description*.

An analytical description represents an algorithm in form of some formulas, e.g., a set of equations, regular expressions, recursive functions, formulas of the λ–calculus etc. Formulas represent relations in sets of processed/computed data. It is an implicit form because it is necessary to derive rules for computation from these formulas.

A prescriptive description provides rules for computation in an explicit form. The most popular examples of this form are computer programs, formal or generative grammars, deduction rules in logical calculi, finite automata, and inference rules in expert systems.

An embodied description provides not only rules for computation but also a description of a device that performs these computations. Usually this form is called an abstract machine or automaton. The most popular examples of this form are different versions of Turing machines, random access machines, Boolean circuits etc.

Any kind of representation of algorithms must include, implicitly or explicitly, *metarules*. These metarules define how an algorithm starts functioning, where from it takes its input, how the rules from the algorithm are applied to data, in what cases the algorithm stops, how and where it gives its output, and what is the result of its functioning. As a result, the same hardware and software allow one to define automata of different types. For example, metarules for result determination can give three different types of automata: *computing*, *accepting* or *deciding*. Thus, we have computing and accepting finite automata, computing, accepting, and deciding Turing machines and so on. The same structure (hardware) of a Turing machine and the same rules (software) allows one to define ordinary Turing machines, inductive Turing machines (Burgin, 1983), infinite time Turing machines (Hamkins and Lewis, 2000) or persistent Turing machines (Goldin and Wegner, 1988).

It is possible to call metarules from an algorithm representation by the name *metasoftware*. Computers also have metasoftware in the form of user instructions. There are many kinds of such instructions: instructions to the computer as a whole, instructions to different program systems and separate programs, instructions to separate devices (printer, scanner etc.) and so on.

We begin with an abstract automaton as the most complete representation of algorithm. The first abstract automaton was Turing machine created by Turing (1936). Von Nemann (1951) introduced a general concept of an abstract automaton. Finite automata were formalized, modified and studied by Mealy (1953), Kleene (1956), and Moore (1956). Potentially infinite automata were formalized, modified and studied by Church (1957).

Abstract automata and algorithms work with symbols, words, and other symbolic configurations, transforming them into other configurations of the same type. For example, words are transformed into words. While words and strings are linear configurations, there are many useful abstract automata and algorithms work with such configurations as graphs, vectors, arrays, etc. (cf., for example, Kolmogorov, 1953; Rabin, 1969; Pratt, Rabin, and Stockmeyer, 1974; Burgin and Karasik, 1975; Burgin, 1982a; Van Leeuwen and Wiedermann, 1985).

An abstract automaton consists of an abstract information processing device (*abstract hardware*), an algorithm/program of its functioning (*abstract software*), and description/specification of information which is processed (*abstract infware*).

In a general case, the hardware of an abstract automaton, consists of three main parts: abstract *input device*, abstract *information processor*, and abstract *output device*, which are presented in Figure 1. For example, Turing machine can have special input and output tapes or one and the same tape works as an input/output device and a part of the processor. Neural networks also have these parts: the input device that comprises all input neurons, output device that consists of all output neurons, and information processor that includes all hidden neurons. In some cases, input and output neurons are combined into one group of visible neurons. In some cases, as for example, for finite automata, input and output devices are not specified.

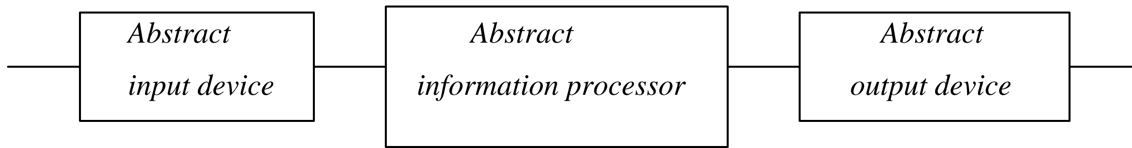


Figure 1. The structure of an abstract automaton

Accordingly, we have a *structural classification* of automata (Fisher, 1965):

1. Abstract automata without input device. They are called *generators*.
2. Abstract automata without output device. They are called *acceptors*.
3. Abstract automata with both input and output devices. They are called *transducers*.

A finite state transducer, for example, is a finite state machine with a read-only input and a write-only output. The input and output cannot be reread or changed. Decision tables (Humby, 1973) represent the simplest case of transducers that have no memory. Basically any abstract automaton can be considered as a transducer.

At the same time, any transducer can also work as an acceptor or as a generator. For example, very often Turing machines are considered as acceptors (cf. Hopckroft *et al*, 2001), although they produce some output, which is written in their tapes at the time when they stop. In some cases, it is assumed that Turing machines always start from the initial state with empty input. It means that they work as generators. However, it is possible to prove that theoretically both only accepting and only generating Turing machines are equivalent with respect to their power to such Turing machines as work as transducers.

Let us consider some examples of different models of algorithms.

Example 2.1. A finite automaton A.

Infware of A is the *linguistic structure* $L = (\Sigma, Q, \Omega)$ in which 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 A.

Hardware of A is the *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 A.

Software or program of A is the *transition function*

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

For a deterministic finite automaton A, δ is a function. For a nondeterministic finite automaton A, δ is a binary relation or correspondence from $\Sigma \times Q$ to $Q \times \Omega$. For an ε-nondeterministic finite automaton A, δ is a binary relation or correspondence from $\Sigma_\varepsilon \times Q$ to $Q \times \Omega$ where $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$.

A finite automaton C without output, or an accepting finite automaton, does not have output symbols in its linguistic structure and therefore its *transition function* is simpler:

$$\delta_C : \Sigma \times Q \to Q$$

Example 2.2. A Turing machine T.

Infware of the Turing machine T consists of languages $\boldsymbol{L} = (L_I, L_W, L_O)$ where L_I is the *input* language, L_W is the *working* language, and L_O is the *output* language of T.

Hardware or device of the Turing machine T. What is the hardware of a computer? It consists of all devices (processor, system of memory, display, keyboard, etc.) that constitute the computer. In a similar way, a Turing machine T has three abstract devices: a finite automaton A, which we may the controller of T and which controls performance of the Turing machine T; a processor or operating device h, which is traditionally called the head of the Turing machine T; and the memory L, which is traditionally called the tape or tapes of the Turing machine T.

Software of the Turing machine T is a single program in a form of simple rules. In the traditional representation, these rules have the following form.

$$q_h a_i \to a_j q_k, \; q_h a_i \to \mathrm{R} q_k, \; q_h a_i \to \mathrm{L} q_k$$

Each rule directs one step of computation of the corresponding Turing machine.

If such rules are given without any specified hardware, we have the second type of algorithms representation: a program. Examples of programs that represent algorithms are computer programs in any programming language: LISP, ALGOL, C^{++}, Java etc.

Analytical repr esentations of algorithms does not contain all information presented in a program. For example, analytical representation does not necessarily include data types. Analytical, prescriptive, and embodied representations are divided into two classes: *formal* and *informal.* Examples of formal analytical representations are formal grammars, primitive

recursive, recursive and partial recursive functions. Examples of informal prescriptive representations are different block schemes and flow charts.

3. Functioning of Automata and Algorithms

Processes represented by algorithms have different types. This implies the corresponding classification of algorithms.

Transformation algorithms describe how to transform some input into a definite output, for example, how to calculate 123 + 321.

Performance algorithms describe some activity or functioning, for example, algorithms of functioning of a car engine or algorithms of human-computer communication. Performance algorithms also describe organized mental activity.

Construction algorithms describe how to build objects.

Decision algorithms are examples of construction algorithms because they describe decision-making and construct decisions.

Algorithms are also classified by objects that are involved in the processes they describe. In such a way, we have:

1. *Material algorithms*, which work with material objects (for example, algorithm of a vending machine or algorithms of pay phone functioning);
2. *Symbolic algorithms*, which work with symbols (for example, computational algorithms are symbolic algorithms that control computing processes);

and

3. *Mixed algorithms*, which work both with material and symbolic objects (for example, algorithms that control production processes or algorithms of robot functioning).

All mathematical models of algorithms are symbolic algorithms. Thus in computer science and mathematics, only symbolic algorithms, i.e., algorithms with symbolic input and output are studied.

Algorithms that work with finite words in some alphabet X are the most popular in the theory of algorithms. As a rule, only finite alphabets are utilized. For example, natural numbers in the decimal form are represented by words in the alphabet $X = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, while in binary form they are represented by words in the alphabet $X = \{0, 1\}$. The

words in X may represent natural numbers or other entities, but in any way there is a natural procedure to enumerate all such words. This makes it possible, when it is necessary, to assume that algorithms work with natural numbers. In such a way, through enumeration of words, any algorithm A defines a partial function $f_A : \mathbf{N} \to \mathbf{N}$ (cf., (Burgin, 1985)). However, there are many reasons to consider algorithms that work with infinite words (Vardi and Wolper, 1994) or with such infinite objects as real numbers (Abramson, 1971; Blum, *et al.* 1998).

Computer algorithms, i.e., such algorithms that are or may be performed by computers, form an important class of all algorithms. Thus, we come to the concept of *computation*. There are two approaches to this concept. According to an engineer, computation is any thing computer can do. On one hand, this restricts computation to computers that exist at a given time. Each new program extends the scope of computation. On the other hand, not everything that computers do is computation. For example, interaction with users or with other computers, sending e-mails, and connecting to the Internet are not computations. Mathematical approach reduces dependence on computers. According to a mathematician, computation is a sequence of symbolic transformations that are performed according to some algorithm. From the mathematical point of view, computers function under the control of algorithms, which are embodied in computer programs. So, to understand possibilities of computers and their boundaries, we have to study algorithms.

To form an algorithm, a system of rules must have a description how these rules are applied. This description consists of metarules for an algorithm, given in a form of some abstract machine. Metarules have often the form of an abstract automaton.

Taking the most general case of automata with input and output devices, we see that there are three main types of automata with respect to their connection with environment.

1. *Reactive automata* that need to be given some input, to produce output.
2. *Proactive automata* that themselves find initial information to produce output.
3. *Interactive automata* can work both as reactive and proactive automata.

Reactive automata represent the classical approach to computation (cf., for example, (Turing, 1936)). Proactive automata represent the intelligent agent approach to computation (cf., for example, (Russel and Norvig, 1995)). Interactive automata represent the interactive approach to computation (cf., for example, (Wegner, 1998)).

The general idea of abstract automaton implies that there are three main modes of processing input data by an abstract or real information processing system:

1. The *computing* mode when an automaton produces (computes or outputs) some words (its output or configuration) as a result of its activity.
2. The *deciding* mode when an automaton, given a word/configuration u and a set X of words/configurations, indicates (decides) whether this word/configuration belongs to X or not.
3. The *accepting* mode when an automaton, given a word/configuration u, either accepts this word/configuration or rejects it.

Remark 3.1. Automata without output device can work only in the accepting mode. Automata without input device can work only in the computing mode. Automata with all three devices can work in all three modes.

In addition, there are two partial deciding modes:

4. The *positive deciding* mode when an automaton, given a word/configuration u and a set X of words/configurations, indicates (decides) whether this word/configuration belongs to X.
5. The *deciding* mode when an automaton, given a word/configuration u and a set X of words/configurations, indicates (decides) whether this word/configuration does not belong to X.

Sometimes (cf., for example, (Hamkins and Lewis, 2000)), the positive deciding mode is called *semi-decidable*.

These types not only reflect the principal modes of computer functioning, but also define the main utilization modes for algorithms and programs. There are several kinds of each mode. For example, there is acceptation by a state and by a result.

Definition 3.1. An automaton A *accepts* a word u *by a result* if A gives some result, or gives a definite result (e.g., the symbol 1 or the word *yes*), when u is its input.

For example, a Turing machine T accepts a word u if and only if: 1) T gives some output when u is its input; after producing the output T halts. At the same time, it is possible to demand that a Turing machine T accepts a word u if and only if the produced result is equal to 1. It is possible to show that both ways of acceptation are equivalent.

Remark 3.2. As a rule it is possible to reduce acceptation by some result to acceptation by a definite result.

To define acceptation by a state, some states of the automaton are defined as final or accepting.

Definition 3.2. An automaton A *accepts* a word u *by a state* if to accept u, A has to come to a final or accepting state when u is given as its input.

Remark 3.3. For many classes of algorithms or abstract automata, acceptance of a word u means that the automaton/algorithm that works with the input u comes to an inner state that is an accepting state for this algorithm or automaton. Finite automata give an example of such a class. However, for such algorithms that produce output, the acceptance assumption means that whenever an algorithm comes to an inner accepting state it produces some chosen result (e.g., the number 1) as its output. In such a way, this algorithm informs that it has reached an inner accepting state.

Another way to define an accepting state is to consider a state of some component of an abstract automaton. For example, pushdown automata accept words not only by an accepting inner state, but also by an empty stack, that is, by a definite state of their stack, which is an external state for these automata.

Definition 3.3. An automaton A *accepts* a word u *by a component state* if a chosen component of A comes to a final or accepting state when u is the input of A.

For a Turing machine, such accepting component is its control device, while for push down automaton such accepting component is either its stack or its control device (Hopcroft *et al*, 2001).

Remark 3.4. For finite automata (Trahtenbrot and Barzdin, 1970) and for pushdown automata (Hopcroft *et al*, 2001), it is proved that acceptance by a result is functionally equivalent to acceptance by a state.

It is possible to prove the following general result.

Theorem 3.1. If a class **K** of automata allows one to enhance each automaton from with an output that informs about the inner state of this automaton, without changing the process (result) of acceptation, then acceptance by a state is reducible to acceptance by a result.

On the other hand, it is possible to do the inverse reduction.

Theorem 3.2. If a class **K** of automata allows one to enhance each automaton from with an output register that stores output, without changing the process (result) of acceptation, then acceptance by a result is reducible to acceptance by a state.

All these forms of acceptation are *static*. At the same time, there are dynamic forms of acceptation. They depend on the behavior of the automaton. For example, a finite automaton *A* accepts infinite strings when *A* comes to an accepting state infinitely many times. Such automata are called Büchi automata (Büchi, 1960). Another example is inductive Turing machine (Burgin, 2003). It produces a result or accepts a word when its output stabilizes.

An important distinction exists between *deterministic* and *non-deterministic* algorithms. At first, such condition as complete determination of each step of an algorithm was considered as necessary for a general model of algorithm. For a long time, all models were strictly deterministic. However, necessity to reflect real situations and model computational processes influenced introduction of non-deterministic algorithms (Rabin and Scott, 1959).

Thus, we may have an impression that the extensive diversity of models results in a similar diversity for the concepts of algorithm, computation and computable function. Nevertheless, mathematicians and computer scientists found that the algorithmic reality is well-organized. They have found a unification law for this reality, which was called the Church-Turing Thesis.

Thus, different definitions of algorithm bring us to the conclusion that algorithms consist of rules or instructions. As a rule, it is supposed that each instruction is performed in one step. This implies several features of algorithms:

1. *Algorithms function in a discrete time.*
2. *All instructions are sufficiently simple.*
3. *Relations between operational steps of algorithm determine topology of computation.*

These properties look very natural. However, some researchers suggest models of computation with continuous time. Examples are given by real number computations in the sense of Shannon (1941) and Moore (1996). Instructions may look simple, but their realization may be very complex. For example, even addition with infinite precision of two transcendental numbers in numerical form is, as a rule, impossible, in spite that its description in an algebraic form is really simple.

Algorithms for computers generate and direct a diversity of computations. These computations have different characteristics. One of the most important of them is the

computation topology. The most popular types of the computation topology with the corresponding computing architecture are:

1. *Sequential computation*.
2. *Parallel or synchronous computation*.
3. *Concurrent or asynchronous computation*.

Each of these types has several subtypes:

1. **Sequential computation** may be:
   1.1. *Acyclic*, i.e., such that it has no cycles;
   1.2. *Cyclic*, i.e., organized in one cycle;
   1.3. *Incyclic*, i.e., it is not one cycle, but contains cycles.
2. **Parallel computation** may be:
   2.1. *Branching*, which means parallel processing of different data from one package of data;
   2.2. *Pipeline*, which means synchronous processing of similar elements from different packages of data (Kogge, 1981);
   2.3. *Extending pipeline*, which combines properties both of branching and pipeline computations (Burgin, Liu, and Karplus, 2001a).
3. According to the control system for computation, **concurrent computation** may be:
   3.1. *Instruction controlled*, which means parallel processing of different data from one package of data;
   3.2. *Data controlled*, which means synchronous processing of similar element from different packages of data;
   3.3. *Agent controlled*, which means that another program controls computation.

While two first approaches are well known and popular, the third type exists without recognition and is not considered as a separate approach. However, even now the third approach is often used implicitly for organization of computation. Examples of agent controlled computations are utilization of an interpreter, which taking instructions of the program, transforms them into machine code, and then this code is executed by the computer. Interpreter is the agent that controls the process. Universal Turing machine is a theoretical example of agent controlled computations. In this case, the program of the

universal Turing machine is the agent that controls the process. In future the role of agent controlled computations will grow immensely.

Usually, it is assumed that algorithms satisfy specific conditions of non-ambiguity, simplicity and effectiveness of separate operations, which have to be organized for an automatic performance. Thus, each operation in an algorithm must be sufficiently clear so that it does not need to be simplified for its performance. Since an algorithm is a collection of rules or instructions, we must know the correct order in which to execute the instructions. If the order is unclear, we may perform the wrong instruction or we may be uncertain which instruction should be performed next. This characteristic is especially important for computers. A computer can only execute an algorithm if it knows the exact order of steps to perform.

Thus, it is assumed traditionally that the principal characteristics of algorithm are:

1) *an algorithm consists of a finite number of rules*;
2) *the rules constituting an algorithm are unambiguous (definite), simple to follow (effective), and have simple finite description (are constructive)*;
3) *an algorithm is applied to some collection of input data and aimed at a solution of some problem*.

This minimal set of properties allows one to consider algorithms from a more general perspective: those that work with real numbers or even with continuous objects; those that do not need to stop to produce a result; and those that use infinite and even continuous time for computation.

We are not going to discuss here what algorithm is or to give a comprehensive definition. Our goal is to find some simple properties of algorithms in general, to present these properties in a form of axioms, and to deduce from these axioms theorems that describe much more profound properties of algorithms. This allows one, taking some class **A** of algorithms, not to prove these theorems but only to check if the initial axioms are valid in **A**. If this is the case, then it makes possible to conclude that all corresponding theorems are true for the class **A**. As we know, computer scientists and mathematicians study and utilize a huge variety of different classes and types of algorithms, automata, and abstract machines. Consequently, such an axiomatic approach allows them to obtain many properties of studied algorithms in a simple and easy way.

4. Basic Axioms for Algorithms

We consider three types of a priori assumptions: *postulates*, *axioms*, and *conditions*.

Postulates are the most basic assumptions. Any class that is considered satisfies, at least, one postulate.

Axioms are global assumptions that represent important properties of algorithms and are used frequently in different situations.

Conditions are local assumptions that represent specific properties of algorithms and are used only for definite results.

The **first group of postulates** defines what algorithms are doing and in what mode abstract automata are functioning.

Let **K** be a class of automata/algorithms that take inputs from a set X and give outputs that belong to a set Y.

Computation Postulate PCM. Any algorithm A from **K** determines a binary relation r_A in the direct product $X \times Y$ of all its inputs X_A and all its outputs Y_A.

Examples of classes that compute relation and not a function are: non-deterministic computing finite automata, algorithms for multiple computations (Burgin, 1983), and interactive algorithms (Wegner, 1998; Van Leeuwen and Wiedermann, 2000).

Definition 4.1. The set X_A is called the domain of A and the set Y_A is called the codomain of A.

Definition 4.2. A relation r in the direct product $X \times Y$ is called computable in **K** if $r = r_A$ for some algorithm A from **K**.

Deterministic Computation Postulate PDC. Any algorithm A from **K** determines a function f_A from X into Y.

Informally it means that given some input, A always produces the same result.

Remark 4.1. Functions may be partial and total. The latter are defined for all elements of X.

Definition 4.3. A function f from X into Y is called computable in **K** if $f = f_A$ for some algorithm A from **K**.

Totality Axiom AT. Any algorithm A from **K** determines a total function f_A from X into Y.

Informally it means that A gives output for any input.

Classes of finite automata and primitive recursive functions satisfy Totality Axiom.

Classes of Turing machines and partial recursive functions do not satisfy Totality Axiom.

This axiom brings us to the distinction between the domain and definability domain of an algorithm.

Definition 4.4. The domain D(A) of an algorithm A is the set X of elements that are processed by A.

Definition 4.5. The definability domain DD(A) of an algorithm A is the set X of elements from D(A) such that if any of them is given as input to A, then A gives a result/output.

This allows us to define domains for a class **K** of algorithms/automata.

Definition 4.6. $D_*(\mathbf{K}) = \cap_{A\in \mathbf{K}} D(A)$ is the lower domain of **K**.

Definition 4.7. $D^*(\mathbf{K}) = \cup_{A\in \mathbf{K}} D(A)$ is the upper domain of **K**.

These concepts make possible to consider classes of algorithms that work with different entities, e.g., numbers, words, graphs, etc.

Definition 4.8. $DD_*(\mathbf{K}) = \cap_{A\in \mathbf{K}} DD(A)$ is the lower definability domain of **K**.

Definition 4.9. $DD^*(\mathbf{K}) = \cup_{A\in \mathbf{K}} DD(A)$ is the upper definability domain of **K**.

Lemma 4.1. $D_*(\mathbf{K}) \subseteq D^*(\mathbf{K})$, $DD_*(\mathbf{K}) \subseteq DD^*(\mathbf{K})$, $DD_*(\mathbf{K}) \subseteq D_*(\mathbf{K})$, and $DD^*(\mathbf{K}) \subseteq D^*(\mathbf{K})$.

Lemma 4.2. If the empty, i.e., nowhere defined, function $f_\varnothing$ is realized by some algorithm from **K**, then $DD_*(\mathbf{K}) = \varnothing$.

Lemma 4.3. If the identity function is realized by some algorithm from **K**, then $DD^*(\mathbf{K}) = D^*(\mathbf{K})$.

In a similar way, we define codomains and ranges for the class **K**.

Definition 4.10. $C_*(\mathbf{K}) = \cap_{A\in \mathbf{K}} C(A)$ is the lower codomain of **K**.

Definition 4.11. $C^*(\mathbf{K}) = \cup_{A\in \mathbf{K}} C(A)$ is the upper codomain of **K**.

Definition 4.12. $R_*(\mathbf{K}) = \cap_{A\in \mathbf{K}} R(A)$ is the lower range of **K**.

Definition 4.13. $R^*(\mathbf{K}) = \cup_{A\in \mathbf{K}} R(A)$ is the upper range of **K**.

Lemma 4.4. $C_*(\mathbf{K}) \subseteq C^*(\mathbf{K})$, $R_*(\mathbf{K}) \subseteq R^*(\mathbf{K})$, $R_*(\mathbf{K}) \subseteq C_*(\mathbf{K})$, and $R^*(\mathbf{K}) \subseteq C^*(\mathbf{K})$.

Lemma 4.5. If the identity function is realized by some algorithm from **K**, then $D^*(\mathbf{K}) = R^*(\mathbf{K})$.

Acceptation Postulate PAM. Any algorithm A from **K** determines a subset $L(A)$ in the set of all its inputs X.

When $X = V^*$ is the set of all words in some alphabet V, then $L(A)$ is called the language of the algorithm A.

Finite automata and Turing machines determine such sets by final states.

Push-down automata determine such sets by final states or by empty stacks.

In general, one of the most popular ways to determine a subset Z of a set X is to define the *characteristic function* c_Z of Z in X. Traditionally (cf., for example, (Fraenkel and Bar-Hillel, 1958)) this function is defined by the following rules:

$c_Z(x) = 1$ when x is an element from Z and $c_Z(x) = 0$ when x does not belong to Z.

A *separation function* is a generalization of the characteristic function. Namely, the separation function s_Z of Z in X is defined by the following rules:

there is an element a in the range of s_Z such that $s_Z(x) = a$ when x is an element from Z and $s_Z(x) = b$ for some $b \neq a$ when x does not belong to Z.

Decision Postulate PDM. Any algorithm A from **K** determines the separation function for a subset $K(A)$ in the set of all its inputs X, giving a fixed output on all elements from $K(A)$ and another output on all elements that do not belong to $K(A)$.

A *dichotomic separation function* of Z in X is a separation function of Z in X that takes one and the same value for all elements that do not belong to Z. A characteristic function is an example of a dichotomic separation function. Usually, computability in a class **K** of a separation function for some set Z implies computability in the class **K** the dichotomic separation function for the same set.

Some classes of algorithms do satisfy PDM although they satisfy its weaker version PSM.

Semidecision Postulate PSM. Any algorithm A from **K** determines the indicator function for a subset $K(A)$ in the set of all its inputs X, giving a fixed output a for all elements from $K(A)$ as inputs and only for them.

Here a function $h(x)$ is called the *indicator function* for a set Z, if there is an element such that $h(x) = a$ if and only if $x \in Z$. An example of an indicator function is the *partial characteristic function* c_Z of Z in X defined by the following rules:

$c_Z(x) = 1$ when x is an element from Z and $c_Z(x)$ is undefined when x does not belong to Z.

Lemma 4.6. PSM implies PAM.

Lemma 4.7. PDM implies PSM if the following condition is satisfied in the class **K**:

(Z) For any algorithm A from **K** and any element a from the set Y_A, there is an algorithm B from **K** such that $B(x) = A(x)$ when $A(x) = a$ and $B(x)$ is undefined when $A(x) \neq a$.

Remark 4.2. This condition is true for many natural classes of algorithms and programs. For instance, when a Turing machine gives some $c \neq a$ as its result, it is possible to change the rules of the machine so that it goes into an infinite cycle after getting c.

Types of algorithms defined by Postulates PCM, PAM, and PDM correspond to the main modes of automaton/algorithm functioning. Given an automaton/algorithm A, we say that A:

computes a set X_A (a formal language L_A) if X_A (correspondingly, L_A) consists of all outputs of A;

accepts a set X_A (a formal language L_A) if X_A (correspondingly, L_A) consists of all elements (words) accepted by A;

decides a set X_A (a formal language L_A) if A, given a word/configuration u, indicates (decides) whether this word/configuration belongs to X_A (to L_A) or not.

It is possible to explain differences in practical meaning of these forms of information processing in the following way. Computability is a way to build objects and to find (compute) their properties. Decidability is a way to find if an object has some property or not. Thus, in a deciding mode, an automaton begins with a property and applies it to an object. In contrast to this, in a computing mode, an automaton begins with an object and derives its properties. Acceptability is a way to select (choose) objects according to their properties.

Definition 4.14. The language L_A is called the *computation* (*acceptation* or *decision*) *language of the algorithm/automaton* A.

Remark 4.3 . Usually, when the mode of computation is fixed, L_A is called simply the language of the automaton/algorithm A. For example, in (Hopcroft *et al*, 2001) only the accepting mode is treated. This makes possible to speak about languages of finite automata, push down automata and Turing machines.

The **second group of postulates** defines objects with which algorithms are working.

Usually algorithms and abstract automata work with words in some alphabet V. An example is Turing machines. We call V the working alphabet of algorithms/automata from a given class. Sometimes it is assumed that they transform natural or whole numbers into natural or whole numbers. An example is recursive functions. These properties are formulated as postulates.

Domain Semiotic Postulate PDS. The domain D(A) of A is the set V^* of all finite words in some alphabet V.

Unrestricted Domain Semiotic Postulate PUDS. The domain D(A) of A is the set V^{**} of all strings, finite and infinite, in some alphabet V.

Domain Numeric Postulate PDN. The domain D(A) of A is a set of natural numbers.

Unrestricted Domain Numeric Postulate PUDN. The domain D(A) of A is a set of numbers.

Codomain Semiotic Postulate PCS. The codomain C(A) of A is the set V^* of all finite words in some alphabet V.

Unrestricted Codomain Semiotic Postulate PUCS. The codomain C(A) of A is the set V^{**} of all strings, finite and infinite, in some alphabet V.

Codomain Numeric Postulate PCN. The codomain C(A) of A is a set of natural numbers.

Unrestricted Codomain Numeric Postulate PUCN. The codomain C(A) of A is a set of numbers.

Remark 4.4. Many algorithms (cf., for example, (Krinitsky, 1977) or (Burgin, 1985)) work with more general entities than words. As an example, it is possible to consider as data such configurations that were utilized by Kolmogorov (1953) in his analysis of the concept of algorithm and construction of the most general definition. Configurations are sets of symbols connected by relations and may be treated as multidimensional words, arrays or hypertexts. Hypertext technology is now very important in information processing both for people and computers (Barrett, 1988; Nielsen, 1990; Landow, 1992). Discrete graphs are also examples of configurations.

The **axioms and conditions from third group** provide means for operations with algorithms and automata. These operations represent transformations, joins, and combinations of programs, computers, and networks. Finding properties of algorithmic operations, we discover regularities

in such areas as software reusability and interoperability, network integration, computer clusterization and reliability.

Definition 4.15. An algorithm C is called a *computing sequential composition* of algorithms A and B if $C(x)$ is defined and equal to $B(A(x))$ when: 1) $A(x)$ is defined and belongs to the domain of B ; 2) $B(A(x))$ is defined. Otherwise, C gives no result being applied to x, i.e., $C(x) = *$.

A computing sequential composition of algorithms A and B is denoted by $A \circ B$.

Remark 4.5. In a general case, a sequential composition of algorithms A and B is not unique. For instance, we can take a Turing machine C that is a sequential composition of two Turing machines T and Q. Then we can rename some inner states of C or add to the rules of C such rules that are never performed. This will give us another Turing machine K that is functionally equivalent to C. Thus, by Definition 4.15, K is also the sequential composition of the same machines T and Q by

Sequential C-Composition Axiom ASC. For any two algorithms A and B from $\mathbf{K}$, their computing sequential composition $A \circ B$ also belongs to $\mathbf{K}$.

Accepting sequential compositions are *organized in a more complicated way.*

Definition 4.16. An algorithm C is called an *accepting sequential composition of the first type* of algorithms A and B when C accepts a word x if and only if there are words u and v such that $x = uv$, A accepts u, and B accepts v.

Accepting sequential compositions of the first type are used in classes of finite automata (cf., for example, (Hopcroft *et al*, 2001)).

Sequential A-Composition Axiom AAC1. For any two algorithms A and B from $\mathbf{K}$, their accepting sequential composition $A \circ B$ of the first type also belongs to $\mathbf{K}$.

Definition 4.17. An algorithms C is called an *accepting sequential composition of the second type* of algorithms A and B when C accepts a word x if and only if A accepts x and B accepts the word z written on the tape of A after A accepts x and stops.

Accepting sequential compositions of the first type are used in classes of Turing machines (cf., for example, (Sipser, 1997) or (Hopcroft *et al*, 2001)).

Sequential A-Composition Axiom AAC2. For any two algorithms A and B from $\mathbf{K}$, their accepting sequential composition $A \circ B$ of the second type also belongs to $\mathbf{K}$.

Remark 4.6. In addition to sequential compositions, there are other kinds of compositions of algorithms. An important class is formed by parallel compositions (cf., for example, (Burgin, 1983)). However, other kinds of compositions are studied elsewhere.

A specific kind of composition is related to universal automata and algorithms, which play an important role in computing and are useful for different purposes. The construction of such automata and algorithms is usually based on some codification (symbolic description) $\mathbf{c}: \mathbf{K} \to X$ of all automata/algorithms in **K**.

Definition 4.18. An automaton/algorithm U is *universal* for the class **K** if given a description $\mathbf{c}(A)$ of an automaton/algorithm A from **K** and some input data x for it, U gives the same result as A for the input x or gives no result when A gives no result for the input x.

In some cases, it is important not only to compute the same function, but also to simulate the behavior of other automata/algorithms. This condition is reflected in the concept of a strictly universal automaton/algorithm.

Definition 4.19. An automaton/algorithm U is *strictly universal* for the class **K** if given a description $\mathbf{c}(A)$ of an automaton/algorithm A from **K** and some input data x for it, U simulates A, working with the same input x, and gives the same result as A or gives no result when A gives no result for the input x.

Universal automata/algorithms combine together input data and automaton/algorithm descriptions like computers combine together input data and programs. Thus, descriptions of the form $\mathbf{c}(A)$ (cf. the Codification Axiom **AC**) play the role of programs for U.

Assuming the Domain Semiotic Postulate **PDS**, we have a set V^* as the domain of all algorithms in **K**. Usually algorithms from **K** are codified in the set V^+ that consists of all finite nonempty words in an alphabet V.

Codification Axiom AC. There is an injective mapping $\mathbf{c} : \mathbf{K} \to V^+$.

The word $\mathbf{c}(A)$ is the "code" or "description" of the algorithm A.

The most popular kind of such a codification is the Godel numbering or enumeration of algorithms or computable functions (cf., for example, (Rogers, 1987)).

It is possible to codify any finite system of symbols by words of equal length in a binary alphabet. This implies the following result.

Proposition 4.1. Axiom AC implies that there is an injective mapping $\mathbf{b} : \mathbf{K} \to \{1, 0\}^+$.

However, to have a codification is not enough because we have to work with it. Thus, we need stronger axioms.

Constructive Codification Axiom CAC. There is an algorithm in **K** that realizes an injective mapping $\mathbf{c} : \mathbf{K} \rightarrow V^+$.

Decidable Codification Axiom DAC. There is a decidable injective mapping $\mathbf{c} : \mathbf{K} \rightarrow V^+$, i.e., there is an algorithm in **K** such that given a word w it decides whether w is a code for some algorithm from **K** and is able to restore this algorithm.

Usually classes of recursive algorithms, such as Turing machines, partial recursive functions or two-register push down automata, satisfy both axioms CAC and DAC. Many classes of subrecursive algorithms, such as recursive functions or polynomial time Turing machines, also satisfy both axioms CAC and DAC. In addition, Many classes of super-recursive algorithms, such as limiting partial recursive functions or inductive Turing machines, also satisfy both axioms CAC and DAC.

Universality Axiom AU. For some coding $\mathbf{c} : \mathbf{K}$ ® V^+, there is a universal algorithm in **K**.

Let the domain X of the algorithms in **K** contains more than one element.

Definition 4.20. An automaton/algorithm A_c is called *shifting* if for any x from X, $A_c(x) \neq x$.

For the future, we will need the following property.

Shifting Condition SH. There is a shifting algorithm in **K**.

The majority of natural classes of algorithms satisfy **SH**. In particular, we have the following result.

Proposition 4.2. Any class **K** of algorithms that contains the class **DFA** of all deterministic finite automata satisfies **SH**.

Let the lower domain $D_*(\mathbf{K})$ of **K** contains more than one element and coincides with the lower codomain $C_*(\mathbf{K})$ of **K**.

Definition 4.21. An automaton/algorithm A_{sw} is called *switching* for x and y from X if $A_{sw}(x) = y$ and $A_{sw}(y) = x$.

Switching Condition SW. For any x and y from X, there is a switching for x and y algorithm in **K**.

Local Switching Condition LSW$_{xy}$. For given x and y from X, there is a switching for x and y algorithm in **K**.

It is possible to realize a switching algorithm by a simple finite automaton. This implies the following result.

Proposition 4.3. Any class **K** of automata/algorithms that includes the class of all finite automata satisfies Condition **SW**.

Condition CA2. The alphabet V that is used for building words processed by algorithms from **K** has, at least, two different elements.

Definition 4.22. An automaton/algorithm A_c is called *weakly switching* for x and y from X if the automaton A_w given the input y gives no output and given the input x, gives y as the output.

Weak Switching Condition WSW. For any x and y from X, there is a weakly switching for x and y algorithm in **K**.

Weak Local Switching Condition WLSW$_{xy}$. For given x and y from X, there is a weakly switching for x and y algorithm in **K**.

A_c-**Composition Condition CSH.** For any algorithm A from **K** and the shifting algorithm A_c, their sequential composition belongs to **K**.

A_{sw}-**Composition Condition CSW.** For any algorithm A from **K** and the switching algorithm A_{sw}, their sequential composition belongs to **K**.

Let us assume that **K** satisfies Postulates **PCM** (or **PAM**), **PDS** (or **PDN**), and **PCS** (or **PCN**), as well as Conditions **SH** and **CSH**.

Theorem 4.1. Axioms **AT** and **AU** are mutually exclusive for the class **K**.

Proof. Let us assume that a class of algorithms **K** also satisfies Axiom **AT** and a universal algorithm U with respect to a coding **c : K ® V^+** exists in **K**. Then U is everywhere defined as **K** satisfies **AT**. Conditions **SH** and **CSH** allow us to build in **K** the algorithm $D = P \circ U \circ A_c$ where P is a pairing algorithm that given a word w produces the pair (w, w) and A_c is the shifting algorithm.

Let us consider words $w = \boldsymbol{c}(D)$ and $x = D(w)$. The word x exists because **K** satisfies **AT**. Then we have $P(w) = (w, w) = (\boldsymbol{c}(D), w)$ and $U(\boldsymbol{c}(D), w) = D(w)$ by the definition of a universal algorithm. At the same time, $A_c(D(w)) \neq D(w)$ and we come to the contradiction that $D(w) = A_c(U(P(w))) = A_c(U(\boldsymbol{c}(D), w)) = A_c(D(w)) \neq D(w)$. This completes the proof of the theorem.

Informally, Theorem 4.1 means that either the class **K** has a universal algorithm but its algorithms are not everywhere defined or they are everywhere defined but the class **K**

does not have a universal algorithm. In other words, universality is incompatible with totality.

Theorem 4.1 implies following results.

Corollary 4.1. In the class of all recursive functions, there are no universal algorithms.

Corollary 4.2. In the class of all primitive recursive functions, there are no universal algorithms.

Corollary 4.3. In the class of all finite automata, there are no universal algorithms.

5. Power of algorithms: comparison and evaluation

Having such a diversity of models for algorithms, we need to compare them. To do this, we introduce such concept as power of an algorithm or of a class of algorithms.

Definition 5.1. Power of an algorithm (of a class of algorithms) is a measure of what this algorithm (algorithms from this class) can do.

Algorithms are used to solve problems. So, it is natural to estimate power by those problems that algorithms can solve.

It is natural to consider two kinds of power for algorithms: absolute that is defined with respect to problems and relative that only compares different algorithms and different classes of algorithms.

Definition 5.2. The *absolute problem solving power* of an algorithm A (of a class $\mathbf{K}$ of algorithms) is the set of all problems solvable by this algorithm (by algorithms from this class).

Usually it is impossible to describe the absolute problem solving power of a complex algorithm or a big algorithmic class. That is why we introduce relative problem solving power of algorithms.

Definition 5.3. The *problem solving power* of an algorithm A (of a class $\mathbf{H}$ of algorithms) is *less than or equal to* (is *cognitively weaker than or equivalent to*) the *problem solving power* of an algorithm B (of a class $\mathbf{K}$ of algorithms) when the algorithm B can solve any problem that A can (any problem that can be solved by a algorithm from $\mathbf{H}$ can be also solved by some algorithm from $\mathbf{K}$). Naturally, the algorithm B is *cognitively stronger than or equivalent to* (has *more than or the same problem solving power*) the algorithm A.

We denote these relation by $A \leq_{ps} B$, $A <_{ps} B$, $A =_{ps} B$, $\mathbf{H} \leq_{ps} \mathbf{K}$, $\mathbf{H} <_{ps} \mathbf{K}$, and $\mathbf{H} =_{ps} \mathbf{K}$.

For instance, if **T** is the class of all deterministic Turing machines and **DFA** is the class of all deterministic finite automata, then $\mathbf{DFA} <_{ps} \mathbf{T}$. If **NFA** is the class of all non-deterministic finite automata, then $\mathbf{NFA} =_{ps} \mathbf{DFA}$.

Different modes of information processing allow one to introduce special concepts: *computing power*, *accepting power*, *decision power*, and *equivalence* of algorithms and their classes.

Definition 5.4. The *computing power* of an algorithm A is *less than or equal to* (is *weaker than or equivalent to*) the *computing power* of an algorithm B when the algorithm B can compute everything that A can compute. Naturally, the algorithm B is *stronger than or equivalent to* (has *more than or the same computing power*) the algorithm A.

Let us take as an example algorithms that solve some algorithmic problem P for a class of algorithms **A**. Such problem P may be: to define whether a given algorithm from **A** gives the result for some data; to define whether a given algorithm from **A** gives the result for all possible data; or to define whether a given algorithm from **A** has more computing power than another given algorithm from **A**.

Proposition 5.1. If an algorithm A solves the problem P for the class $\mathbf{A}_1$, an algorithm B solves the problem P for the class $\mathbf{A}_2$, and $\mathbf{A}_1 \subset \mathbf{A}_2$, then the computing power of B is larger than the computing power of A.

Definition 5.5. Two algorithms A and B are *functionally equivalent* (or simply, *equivalent*) if the algorithm B can compute everything that A can compute and the algorithm A can compute everything that B can compute.

Taking into account different modes of algorithm functioning, we come to a more general concept.

Definition 5.6. Two algorithms are called *functionally equivalent with respect to computability* (*acceptability*, *positive decidability*, *negative decidability* or *decidability*) if they define in the corresponding mode the same function f_A or relation r_A.

Example 5.1. In the theory of finite automata, functional equivalence means that two finite automata accept the same language (Hopcroft *et al*, 2001). This relation is used frequently to obtain different properties of finite automata. The same is true for the theory of pushdown automata.

However, in general, functional equivalence is stronger than linguistic equivalence defined below.

Definition 5.7. Two algorithms are called *linguistically equivalent with respect to computability* (*acceptability*, *positive decidability*, *negative decidability* or *decidability*) if they define in the corresponding mode the same language L_A .

As L_A is the range of the function f_A (relation r_A), we have the following result.

Proposition 5.1. Two algorithms are linguistically equivalent with respect to computability (acceptability, positive decidability, negative decidability or decidability) if they are functionally equivalent with respect to computability (acceptability, positive decidability, negative decidability or decidability).

The inverse statement is not true in general as the following example demonstrates.

Example 5.2. Let algorithm A computes the function $f_A(n) = n + 1$ for all $n = 1, 2, 3, \ldots$ and E is an identity algorithm on the set $\{2, 3, \ldots \}$, i.e., $E(m) = m$ for all $n = 2, 3, \ldots$. Then A and E are linguistically equivalent with respect to computability, but they are not functionally equivalent with respect to computability.

While in practice, we usually compare individual algorithms, for theory it is even more important to compare power of different classes of algorithms.

Definition 5.8. A class of algorithms $\boldsymbol{A}$ has *less or equal computing power* than (is *weaker than or equivalent to*) a class of algorithms $\boldsymbol{B}$ when algorithms from $\boldsymbol{B}$ can compute everything that algorithms from $\boldsymbol{A}$ can compute. Naturally, the class of algorithms $\boldsymbol{B}$ is *stronger than or equivalent to* (has *more or equal computing power than*) the class of algorithms $\boldsymbol{A}$.

Remark 5.2. Any mathematical or programming model of algorithms defines some class of algorithms. Thus, comparing classes corresponding to models, we are able to compare power of these models.

For example, the class of all finite automata is weaker than the class of all Turing machines. It means that a Turing machine can compute everything that can compute finite automata. However, there are such functions that are computable by Turing machines while finite automata cannot compute them.

Remark 5.3. Definitions 5.2 - 5.4 and 5.8 are not exact because they contain such a term as *everything*, which is very imprecise. To develop a mathematical theory of algorithms and automata, we need completely formal constructions. Formalization is achieved in different ways.

The first one is to relate to an algorithm the function that it computes. Such formalization is done in Definitions 5.5, 5.7, 510, and 5.11. The second approach is based on introduction of sets or languages defined by an automaton/algorithm and consideration of three modes of automaton functioning: *computation*, *acceptation*, and *decision*. Such formalization is done in Definitions 5.6 and 5.9.

As it is known, any set of words form a *formal language*. This allows us to cons ider full output of algorithms, that is, the set L_A of all words that an algorithm A computes, accepts or decides.

This allows us to compare computing, accepting and deciding power of algorithms.

Definition 5.9. A class of algorithms $\boldsymbol{A}$ has *less or equal linguistic computing* (*accepting or decision*) *power* than (is *linguistically weaker than or equivalent to*) a class of algorithms $\boldsymbol{B}$ when algorithms from $\boldsymbol{B}$ can compute (accept or decide, correspondingly) any language that algorithms from $\boldsymbol{A}$ can compute (accept or decide). Naturally, the class of algorithms $\boldsymbol{B}$ is *linguistically stronger than or linguistically equivalent to* (has *more or equal linguistic computing* (*accepting or decision*) *power than*) the class of algorithms $\boldsymbol{A}$.

Instead of using sets or languages, it is possible to use functions for comparison of computing power.

Definition 5.10. A class of algorithms $\boldsymbol{A}$ has *less or equal functional computing power* than (is *functionally weaker than or equivalent to*) a class of algorithms $\boldsymbol{B}$ when algorithms from $\boldsymbol{B}$ can compute any function that algorithms from $\boldsymbol{A}$ can compute.

Definition 5.11. Two classes of algorithms are *functionally equivalent* (or simply, *equivalent*) if they compute the same class of functions, or relations for non-deterministic algorithms.

Remark 5.7. Lin guistic equivalence of algorithms is stronger than functional equivalence because more algorithms are glued together by linguistic equivalence than by functional equivalence. Really, when two algorithms compute the same function, then they compute the same language. However, it is possible to compute the same language by computing different functions. For example, let us take the alphabet $\{a, b\}$ and two automata A and B. The first one gives as its output the word that is its input. It computes the identity function $f(x) = x$. The automaton B gives the following outputs: $B(\varepsilon) = \varepsilon$, $B(a) = b$, and $B(b) = a$. As a result, A and B are equivalent, but not functionally equivalent.

6. Problems and Related Properties of Algorithms

There are three main types of problems: *cognitive*, *constructive*, and *sustaining* or *preserational*.

A *cognitive* problem is aimed at knowledge acquisition.

A *constructive* problem is aimed at building, transforming or finding some thing.

A *sustaining* problem is aimed at preserving something (a process, data, knowledge, environment, etc.).

To define an algorithmic problem, we at first, consider parametric or class problems in classes of objects. Let us consider a class of objects X.

Definition 6.1. A *parametric* or *class problem* in a class X asks about some property (properties) of an arbitrary object from X. Such problems have one parameter with X as a range and are called unary parametric problems.

For instance, when X is a set of computers, we have following parametric problems:

What is system memory capacity for a given computer c from X ?

What is hard disk capacity for a given computer c from X ?

What is processor clock speed for a given computer c from X ?

It is possible to consider problems with several parameters that have some sets $X_1 , X_2 , \dots ,$ X_n as their ranges.

Definition 6.2. A *parametric problem* in classes $X_1 , X_2 , \dots , X_n$ asks about a relation(s) between arbitrary objects $a_1 , a_2 , \dots , a_n$ from $X_1 , X_2 , \dots , X_n$, correspondingly. Such problems are called n-ary parametric problems.

As a rule, we are interested not in solving a pararmetric problem for one value of the parameter, but in finding a possibility to solve this problem for all values of its parameters. Algorithms can provide such a possibility.

Definition 6.3. An *algorithmic* problem (in classes $X_1 , X_2 , \dots , X_n$) asks to find an algorithm for solving some a parametric problem (in these classes) for all values of its parameters.

Remark 6.1. There are parametric problems that are not algorithmic. For example, the following problems are parametric in the class X of all real continuous functions:

Find a solution of the differential equation $\mathrm{d}u/\mathrm{d}x = f(x)$ in the class $\mathbf{C}(\boldsymbol{R}, \boldsymbol{R})$ of all real continuous functions.

Find necessary and sufficient conditions for a matrix A to be invertible in the class $\mathbf{M}_n$ of all real n-dimensional matrices.

Find the mass of such subatomic particle as neutrino. Because there are many neutrino, this is also a parametric problem.

However, actually it is possible to correspond to any problem an algorithmic one. Indeed, if we have a problem A, then we can correspond to it an algorithmic problem "Find an algorithm that solves A."

Here we are mostly interested not only in finding an algorithm solving some problem, but in having a good or, at least, reasonable algorithm for this purpose. Thus, we study her algorithmic problems with respect to chosen classes of algorithms, abstract automata, formal languages, and programs.

Definition 6.4. An *algorithmic* problem (in classes $X_1, X_2, \dots, X_n$) with respect to a class $\mathbf{K}$ asks to find an algorithm from $\mathbf{K}$ for solving some a parametric problem (in these classes) for all values of its parameters.

Let us consider some important algorithmic problems in a class $\mathbf{K}$ of algorithms with the input domain X and output domain Y.

Definability Problem $\mathbf{R_D}$: given an arbitrary algorithm A from $\mathbf{K}$ and an arbitrary element x from X, find if $A(x)$ is defined.

Local Definability Problem $\mathbf{R_{DA}}$: for a fixed algorithm A from $\mathbf{K}$ and an arbitrary element x from X, find if $A(x)$ is defined.

Fixed Output Problem $\mathbf{R_{O\mathit{b}}}$: given an arbitrary algorithm A from $\mathbf{K}$ and an arbitrary element x from X, find for a fixed element b from Y if $A(x) = b$.

Local Output Problem $\mathbf{L_{O\mathit{b}}}$: given an arbitrary element x from X, find for a fixed algorithm A from $\mathbf{K}$ and a fixed element b from Y if $A(x) = b$.

Free Output Problem $\mathbf{F_{O\mathit{b}}}$: given an arbitrary algorithm A from $\mathbf{K}$, an arbitrary element b from Y, and an arbitrary element x from X, find if $A(x) = b$.

Lemma 6.1. Decidability of $\mathbf{R_D}$ in $\mathbf{K}$ implies decidability of $\mathbf{R_{DA}}$ in $\mathbf{K}$.

Lemma 6.2. Decidability of $\mathbf{F_{O\mathit{b}}}$ in $\mathbf{K}$ implies decidability of $\mathbf{R_{O\mathit{b}}}$ in $\mathbf{K}$ and decidability of $\mathbf{R_{O\mathit{b}}}$ in $\mathbf{K}$ implies decidability of $\mathbf{L_{O\mathit{b}}}$ in $\mathbf{K}$.

Lemma 6.3. If Y is finite, then decidability of $\mathbf{R_{DA}}$ in $\mathbf{K}$ implies decidability of $\mathbf{R_D}$ in $\mathbf{K}$.

Many problems are related to some properties. If we have some property **P**, then we can define two mass problems:

1. The problem $\underline{\mathbf{P}}$ that demands to find if an arbitrary object from the property domain $D(\mathbf{P})$ has this property **P** or not.
2. The problem $\overline{\mathbf{P}}$ that demands to find if an arbitrary object from the property domain $D(\mathbf{P})$ has this property **P**.

The difference between problems $\underline{\mathbf{P}}$ and $\overline{\mathbf{P}}$ is that $\underline{\mathbf{P}}$ demands to give an answer in both cases - when an object has the property **P** and when an object does not have the property **P**, while in $\overline{\mathbf{P}}$ we are interested only in objects that have this property.

In addition to properties of algorithms/automata, there are relations between algorithms/automata. However, relations are kinds of properties (Burgin, 1984). Binary relations can be viewed as properties of pairs. Ternary relations can be viewed as properties of triples and so on. Thus, algorithmic problems are corresponded not only to properties but also to relations. For instance, if we have a binary relation R, then we correspond to it two problems: $\underline{\mathbf{P}}$ and $\overline{\mathbf{P}}$. The first one $\underline{\mathbf{P}}$ demands to find if a given pair belongs to R or not, while the second problem $\overline{\mathbf{P}}$ demands to find if a given pair belongs to R.

Definition 6.5. A binary relation is called trivial if it is empty or contains all pairs from its domain.

At the same time, relations induce many properties in the same domain. For instance, any binary relation R in **K** for each A from **K** determines the property R_A = "B has the property R_A if the pair (A, B) belongs to R." This property R_A is called a *section of the relation R*. Sections allow us to prove different decidability results.

Definition 6.6. A property is called trivial if there are no objects with this property or all objects from its domain have this property.

Lemma 6.1. A binary relation is nontrivial if and only if it has some nontrivial section.

Let **H** be a class of algorithms.

Proposition 6.1. If a binary relation R has an undecidable in **H** section, then the relation R is also decidable in **H**.

Proposition 6.2. Any section of a decidable in **H** relation is also decidable in **H**.

Definition 6.7. A mass/parametric problem **Q** is *solvable* in a class **K** of algorithms if there is such an algorithm $R_{\mathbf{P}}$ such that gives a solution to this problem for values of the problem parameters.

Solvability of problems in many cases is related to decidability of properties.

Definition 6.8. A property **P** is *decidable* in a class **K** of algorithms if there is an algorithm $R_{\mathbf{P}}$ such that it defines if an arbitrary object (from X) has the property **P** or not when this object or its description is given to $R_{\mathbf{P}}$ as input.

Let a class **K** satisfies the **Local Switching Condition LSW**$_{yn}$ **.**

Proposition 6.3. A property **P** is decidable in **K** if and only if the property $\neg$**P** is also decidable in **K**.

Definition 6.9. A property **P** is *semidecidable* in a class **K** of algorithms if there is such an algorithm $R_{\mathbf{P}}$ such that it defines if an arbitrary object (from X) has the property **P** when this object or its description is given to $R_{\mathbf{P}}$ as input.

For many classes of algorithms, there are more semidecidable properties than decidable. For instance, halting property of Turing machines is semidecidable but not decidable in the class of all Turing machines.

Proposition 6.4. A mass problem $\underline{\mathbf{P}}$ with the domain X is solvable in a class **K** of algorithms if the corresponding property **P** is decidable.

Corollary 6.1. A mass problem $\underline{\mathbf{P}}$ is solvable in **K** if and only if the mass problem $\underline{\neg\mathbf{P}}$ is also solvable in **K**

Proposition 6.5. A mass problem $\overline{\mathbf{P}}$ with the domain X is solvable in a class **K** of algorithms if the corresponding property **P** is semidecidable.

For many classes of algorithms, there are more semidecidable properties than decidable. It is known for many classes of recursive algorithms, such as Turing machines. In a general setting, we will see this in the next section.

When we consider such objects as abstract automata and algorithms, we come to specific classes of properties.

Definition 6.10. A property P of algorithms from **K** is called *linguistic* if for any two automata/algorithms A and B, $L(A) = L(B)$ implies $P(A) = P(B)$.

Linguistic Properties:

For an automaton/algorithm B:

$$L(B) = \varnothing;$$

$$L(B) \neq \varnothing;$$

$$b \in L(B);$$

$$b \notin L(B).$$

For two automata/algorithms A and B:

$$L(A) = L(B);$$

$$L(A) \subseteq L(B);$$

$$L(A) \cap L(B) = \varnothing;$$

$$L(A) \cap L(B) \neq \varnothing.$$

Linguistic Problems:

Linguistic Equivalence Problem $\mathbf{R_{LE}}$: given two arbitrary algorithms A and B from **K**, find if $L(A) = L(B)$.

Linguistic Inclusion Problem $\mathbf{R_{LI}}$: given two arbitrary algorithms A and B from **K**, find if $L(A) \subseteq L(B)$.

Linguistic Membership Problem $\mathbf{R_{LM}}$: given an arbitrary algorithm A from **K** and an arbitrary element b from Y, find if $b \in L(B)$.

Linguistic Non-membership Problem $\mathbf{R_{LN}}$: given an arbitrary algorithm A from **K** and an arbitrary element b from Y, find if $b \notin L(B)$.

Remark 6.2. It is possible to consider linguistic properties of algorithms as properties of languages determined by algorithms.

Linguistic properties and problems are the most popular in the theory of algorithms because algorithms and automata are usually related to formal languages (cf., for example, (Rogers, 1987) or (Hopckroft *et al*, 2001)). However, there are other important properties of algorithms, programs, and automata.

Definition 6.11. A property P of algorithms from **K** is called *functional* if for any two automata/algorithms A and B, $A_f = B_f$ implies $P(A) = P(B)$.

Functional Properties:

For an automaton/algorithm B:

1. $B_f = f\varnothing$;
2. $B_f \neq f\varnothing$;

3. $DD(B_f) = V^*$;
4. $DD(B_f) = N$.

For two automata/algorithms A and B:

1. Algorithms A and B compute the same function, i.e., $A_f = B_f$;
2. $\forall x\, (A_f(x) \leq B_f(x))$.
3. Algorithms A and B have the same definability domain, i.e., $DD(A) = DD(B)$.
4. $DD(A_f) \cap DD(B_f) = \varnothing$.
5. $DD(A_f) \cap DD(B_f) \neq \varnothing$.

Functional Problems:

1. Find if algorithms A and B have the same characteristic function of the definability domain.
2. Find if algorithms A and B have the same semicharacteristic function of the definability domain.
3. Functional Equivalence Problem $\mathbf{R_{FE}}$: given two arbitrary deterministic algorithms A and B from $\mathbf{K}$, find if algorithms A and B compute the same function, i.e., $f_A = f_B$.

Remark 6.3. It is possible to consider functional properties of algorithms as properties of functions determined by algorithms.

Sometimes to demands that all results of algorithms coincide, as we do in the definition of a functional property, is too much. Thus, we introduce weak functional properties.

Let Z be a subset of the lower definability domain $DD_*(\mathbf{K})$ of $\mathbf{K}$.

Definition 6.12. A property P of algorithms from $\mathbf{K}$ is called *weak functional* on Z if for any two automata/algorithms A and B, $P(A) = P(B)$ when the equality $A_f(x) = B_f(x)$ is true for all for all x from Z.

Weak Functional Properties:

1. Self-application property:

$$SAP(A) = \begin{cases} 1 & \text{if being applied to } \mathbf{c}(A)\text{, the algorithm } A \text{ gives the result;} \\ 0 & \text{otherwise.} \end{cases}$$

2. Self-acceptation property:

$$\mathrm{SAC}(A) = \begin{cases} 1 & \text{if being applied to } \mathbf{c}(A), \text{ the algorithm } A \text{ accepts } \mathbf{c}(A); \\ 0 & \text{otherwise.} \end{cases}$$

3. Self-approval property:

$$\mathrm{SAR}(A) = \begin{cases} 1 & \text{if being applied to } \mathbf{c}(A), \text{ the algorithm } A \text{ gives the result } yes; \\ 0 & \text{if being applied to } \mathbf{c}(A), \text{ the algorithm } A \text{ gives the result } no. \end{cases}$$

By definition, any functional property is a weak functional property. It implies the following result.

Proposition 6.6. If for some subset Z of the lower definability domain DD*(**K**), all weak functional properties of algorithms/automata from **K** are decidable (undecidable, semidecidable) in **K**, then all functional properties of algorithms/automata from **K** are decidable (undecidable or semidecidable, correspondingly) in **K**.

Comparing functional and linguistic properties, we find that there is a strict correspondence between properties of both kinds. To build it, we need additional axioms and conditions.

Condition CD. C(**K**) = D(**K**).

Axiom of a pair correspondence APC. There is an injection p: D(**K**)×D(**K**) → D(**K**)

Axiom of a computable pair correspondence ACPC. There is an injection p : D(**K**)×D(**K**) → D(**K**) such that $p(R_f)$ is computable in **K** for any computable in **K** function f.

Axiom of a decidable pair correspondence ADPC. There is a decidable injection p: D(**K**)×D(**K**) → D(**K**)

Remark 6.4. In conventional set theories, such as ZF (cf. Fraenkel and Bar-Hillel, 1958), **APC** holds if and only if D(**K**) is an infinite set. As the majority of computational models works with inputs from infinite domains, usually, words or

numbers, **APC** is true for almost all conventional models. For classes of recursive algorithms, such as Turing machines or partial recursive functions, these injections are decidable and thus, computable.

Axiom CL. For any algorithm/automaton A from **K**, the language L_A computed by A is equal to the image Im f_A

Let LP(**K**) and FP(**K**) be classes of all linguistic and functional properties of algorithms/automata from **K**, correspondingly.

Theorem 6.1. If the Axiom **ACPC** is true, then there are natural injections **rel**: FP(**K**) $\rightarrow$ LP(**K**) and **ch**: LP(**K**) $\rightarrow$ FP(**K**).

<u>Proof</u>. At first, we show that taking an arbitrary function f: D(**K**) $\rightarrow$ D(**K**) , it is possible to correspond a subset L_f of the set D(**K**) to it so that if g: D(**K**) $\rightarrow$ D(**K**) is another function, then $f = g$ if and only if $L_f = L_g$. Really, there are defined in a standard way binary relations R_f , R_g $\subseteq$ D(**K**)$\times$D(**K**) such that $f = g$ if and only if $R_f = R_g$. Then we put $L_f = p(R_f)$ and $L_g = p(R_g)$ where p is given in the Axiom **ACPC**. L_f and L_g are formal languages and $f = g$ if and only if $L_f = L_g$ because p is an injection.

For a property P of algorithms from **K**, we define **rel**(P) = { H; $H \in P$ and H computes the language $L_H = p(R_f)$ for the function $f = F_A$ some $A \in P$ }. Then if P is a functional property of algorithms from **K**, **rel**(P) is a linguistic property. Indeed, let us consider two algorithms H and F from **K** and assume that H has the property **rel**(P) and they both compute the same language L. It means that $L_H = L_F$. At the same time, $L_H = p(R_f)$ for the function $f = F_A$ some $A \in P$. Consequently, $L_F = p(R_f)$ for the function $f = F_A$ some $A \in P$ { A ; $A \in P$} and by the definition F also has the property **rel**(P). Thus, **rel**(P) is a linguistic property.

Let us take as **ch** the identity mapping, i.e., **ch**(Q) = Q for any property Q, and consider some linguistic property P of algorithms from **K**. We show that P is also a functional property. Indeed, let us consider two algorithms A and B from **K** and assume that A has the property P and they both realize the same function f. We know by the Axiom **CL** that the language L_A computed by A is equal to the image Im f_A and the language L_B computed by B is equal to the image Im f_B . As $f_A = f_B$, Im f_A = Im f_B. As P is a linguistic property P, B also has the property P. As A and B are arbitrary algorithms from **K**, P is a functional property.

Corollary 6.2. Any linguistic property is a functional property.

Remark 6.5. There are functional properties that are not linguistic. To show this let us consider the following property P_0 of Turing machines with the alphabet {1, 0}:

A Turing machine does not give the result for the input 0.

It is possible to define a Turing machine T that realizes the function f such that $f(0w) = w$ and $f(1w) = w$ for any nonempty word w, while $f(0w)$ and $f(1)$ are undefined. If E is a Turing machine that realizes the identity function $e(x) = x$, then $f = f_T \neq f_E = e(x)$ and $L = L$. By the definition, the Turing machine T has the property P_0 , while the Turing machine E does not have this property. Consequently, P_0 is not a linguistic property.

Correspondence between functional and linguistic properties allows us to establish connections between decidability of these properties.

Corollary 6.3. If all functional properties of algorithms/automata from **K** are decidable (undecidable, semidecidable), then all linguistic properties of algorithms/automata from **K** are decidable (undecidable or semidecidable, correspondingly).

Corollary 6.4. If not all linguistic properties of algorithms/automata from **K** are decidable (undecidable, semidecidable), then not all functional properties of algorithms/automata from **K** are decidable (undecidable or semidecidable, correspondingly).

If A is an algorithm/automaton, then $\mathrm{Tr}_A(x)$ denotes the computational trajectory of A generated by an input x. Let Z be a subset of the lower definability domain $DD_*(\mathbf{K})$ of **K**.

Definition 6.13. A property P of algorithms from **K** is called a *performance property* (on Z) if the equality $\mathrm{Tr}_A(x) = \mathrm{Tr}_B(x)$ for all x (from Z) implies $P(A) = P(B)$.

Performance Properties:

1. Given input x, a Turing machine T from a class **T** starts from an empty cell.
2. Given input x, the head of a Turing machine T comes into the cell with number n.
3. Given input x, a Turing machine T uses only $(n-1)$ cells for its computation.
4. Given input x, the head of a Turing machine T comes two times into the cell with number n.
5. A Turing machine T never stops.

Performance Problems:

1. Find if starting with input x, the head of a Turing machine T comes into the cell with number n.
2. Find if a Turing machine T from a class **T** starts from an empty cell.
3. Find if starting with input x, the head of a Turing machine T comes two times into the cell with number n.
4. Find if starting with input x, a Turing machine T from a class **T** never stops.

Definition 6.14. A property P of algorithms from **K** is called a *weak performance property* (on Z) if the equivalence $\mathrm{Tr}_A(x) \sim \mathrm{Tr}_B(x)$ for all x (from Z) implies $P(A) = P(B)$.

For instance, we assume that two trajectories are equivalent when either they are both infinite or they both bring automata to the same output result. In the case of Turing machines that either give no final result or stop functioning in the same final state, this equivalence coincides with the functional equivalence.

Remark 6.6. For non-deterministic automata, one input can result in many (sometimes even infinitely many) different trajectories. Thus, for a definition of performance properties, it is natural to consider systems of trajectories instead of separate trajectories in the case of non-deterministic automata.

If A is an algorithm/automaton, then $\mathrm{Op}_A(q, x)$ denotes the operation that A performs having input x, being in a state q. For automata with memory, the state includes the state of memory.

Definition 6.15. A property P of algorithms from **K** is called *operational* the equality $\mathrm{Op}_A(q, x) = \mathrm{Op}_B(q, x)$ for all inputs x and all states q implies $P(A) = P(B)$.

Operational Properties:

1. The head of a Turing machine moves only to the right for a given input.
2. The head of a Turing machine returns into the first cell for a given input.
3. The head of a Turing machine comes to the n-th cell, at least, two times for a given input.
4. The head of a Turing machine comes to the n-th cell for a given input.
5. The head of a Turing machine moves only to the right for all inputs.
6. The head of a Turing machine returns into the first cell for all inputs.
7. The head of a Turing machine comes to the n-th cell, at least, two times for all inputs.

Operational Problems:

1. Find if starting with input x, the head of a given Turing machine moves only to the right.
2. Find if starting with input x, the head of a given Turing machine returns into the first cell.
3. Find if starting with input x, the head of a given Turing machine returns into the first cell.
4. Find if starting with input x, the head of a given Turing machine comes to the n-th cell.

Proposition 6.6. Any operational property is a performance property.

Remark 6.7. Not all performance properties are operational properties as the following example shows.

Example 6.2. Let us take the class $\mathbf{T}_{St}$ of all automata with the same structure as Turing machines with one linear tape and one head, in which all machines are enumerated and the n-th machine starts from the n-th cell in the tape. It is possible to consider the following property P_x:

"Given an input x, a machine A from $\mathbf{T}_{St}$ starts from an empty cell."

This is a performance property as the equality $\mathrm{Tr}_A(x) = \mathrm{Tr}_B(x)$ means that given the input x, A and B start functioning with the head in the same cell. At the same time, P is not an operational property. Indeed, there are machines A and B in $\mathbf{T}_{St}$ such that have the same active performance rules, but different numbers because one of them (say A) has many passive rules, i.e., rules that are never used. Consequently, these machines start functioning with their heads in different cells. Thus, there is an input x such that A starts from an empty cell and B from a cell with some symbol. As a result, $P_x(A) \neq P_x(B)$, meaning that P_x is not an operational property.

Correspondence between performance and operational properties allows us to establish connections between decidability of these properties.

Corollary 6.5. If all operational properties of algorithms/automata from **K** are decidable (undecidable), then all performance properties of algorithms/automata from **K** are decidable (undecidable).

Corollary 6.6. If not all performance properties of algorithms/automata from **K** are decidable (undecidable), then not all operational properties of algorithms/automata from **K** are decidable (undecidable).

Proposition 6.7. Any performance property is an operational property if all algorithms from **K** start functioning from the same state q_0, given the same input x.

Corollary 6.7. For the class **T** of all Turing machines, performance and operational properties are the same.

Corollary 6.8. For the class **T** of all deterministic finite automata, performance and operational properties are the same.

Corollary 6.9. All operational properties of algorithms/automata from **K** are decidable (undecidable) if and only if all performance properties of algorithms/automata from **K** are decidable (undecidable).

Definition 6.16. Descriptive properties are properties of algorithm descriptions.

Descriptive Properties:

1. The description d(A) of an algorithm A contains the letter "a".
2. A given description d(A) of an algorithm A is minimal for all algorithms that compute the same function.
3. A given description d(A) of an algorithm A is minimal for all algorithms that compute the word x without input.

Descriptive Problems:

1. Does the description d(A) of an algorithm A contain the letter "a"?
2. Is a given description d(A) of an algorithm A is minimal for all algorithms that compute the same function?
3. Is a given description d(A) of an algorithm A is minimal for all algorithms that compute the word x without input?
4. What is the length of the description d(A) of a minimal algorithm A that computes the word x without input?

Remark 6.8. There are properties of algorithms that belong to several types. For instance, the following property P_{dnt}:

"A Turing machine T computes a total function and has a description d(T) with the length less than n,"

is at the same time functional and descriptive.

7. Boundaries for Algorithms and Computation

In this section, we determine what problems are algorithmically solvable and what problems are not.

Condition E. The class **K** contains an algorithm E such that for any input, it gives 1 as its output.

Condition CE. For all automata A in **K**, a sequential composition $A \circ E$ is a member of **K**.

Condition UC. If U is a universal in **K** algorithm, then for any automaton A in **K**, sequential compositions $U \circ A$ and $A \circ U$ are members of **K**.

Theorem 7.1. If a class **K** satisfies Axioms **AU** and **CAC**, and Conditions **E** and **CE**, then the problem $\mathbf{R_D}$ is semidecidable in **K**.

Proof. Let U be a universal in **K** algorithm and C be a coding algorithm for **K**, i.e., $C_f : \mathbf{K} \to V^+$. Then the algorithm R defined by the sequential composition $C \circ U \circ E$ belongs to **K**. This algorithm works in the following manner. Given a pair (x, A) with $x \in V^*$ and $A \in \mathbf{K}$, R codifies A by means of C and then sends the pair $(x, c(A))$ to U. The algorithm U simulates functioning of A with input x. When being applied to x, A gives the result, this result goes to E and E outputs 1. Otherwise, R produces no result. This means that R computes the semicharacteristic function for all positive solutions of the problem $\mathbf{R_D}$. Consequently, $\mathbf{R_D}$ is semidecidable in **K**.

Theorem 7.1 and Lemma 6.1 imply the following result.

Theorem 7.2. If a class **K** satisfies Axioms **AU** and **CAC**, and Conditions **E** and **CE**, then the problem $\mathbf{R_{DA}}$ is semidecidable in **K**.

However, as it is demonstrated in Theorems 7.4 and 7.5, problems $\mathbf{R_D}$ and $\mathbf{R_{DA}}$ are undecidable.

Condition CA. **K** contains a converting algorithm A_C such that it checks whether a given word w is equal to $\mathbf{c}(A)$ for some A in **K** and then if this is true it converts w to $(\mathbf{c}(A), w)$.

Condition CS. For all automata A in **K**, the sequential composition $A \circ A_C$ is a member of **K**.

Condition CC. For all automata A in **K**, the sequential composition $A_C \circ A$ is a member of **K**.

Remark 7.1. Conditions **CS** and **CC** follow from Axiom **ASC**. However, for some classes Condition **CS** may be true, while Axiom **ASC** is invalid. Inductive Turing machines of the first order (Burgin, 2003) is an example of such a class. This class does not include all sequential compositions of its members, but has all sequential compositions of its members with finite automata, while it is possible to realize algorithms A_C and C by finite automata.

Let us consider a class **A** of algorithms/automata that satisfies postulates **PDC**, **PDS**, and **PCS** (or **PDN**, and **PCN**), axioms **AT**, **AC**, and conditions **CA2**, $\mathbf{LSW}_{yn}$ (or **SW**), **CA**, **CC**, and **CS** (or axiom **ASC**).

Remark 7.2. As the working alphabet V of algorithms from **A** contains more than one element, by renaming, we can assume that $\boldsymbol{y}$ and $\boldsymbol{n}$ are elements of V.

Theorem 7.3. The Fixed Output Problem $\mathbf{R}_{\mathbf{O}y}$ is unsolvable in **A**, i.e., there is no automaton H in **A** such that for all algorithms A in **A,** and for all strings w in V^*, this automaton H can check if $A(w) = y$ or not.

Proof. Let us assume that such as automaton H exists in **A**. It means that given a pair $(\mathbf{c}(A), w)$ where $\mathbf{c}: \mathbf{K} \rightarrow \mathrm{V}^+$ is defined by the axiom **AC**, H produces y when $A(w) = y$ and produces n otherwise. Properties of the class **A** provide us with the automata A_C and A_{sw}, and allow us to build the sequential composition $B = A_C \circ H \circ A_{sw}$, which also belongs to **A** and in which the automaton A_{sw} switches y and n.

This composition B is presented in Figure 2.

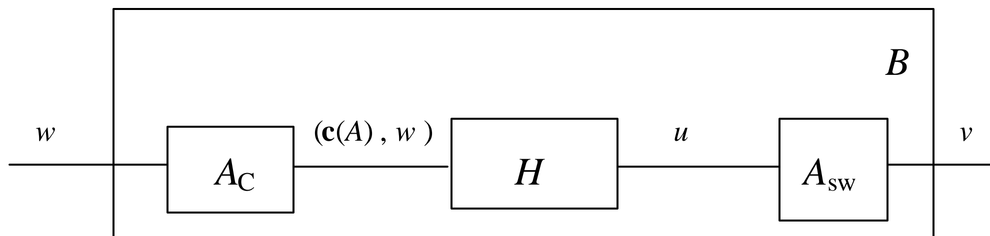


Figure 2. A hypothetical automaton B

Let us consider functioning of the automaton B when it works with the word $\mathbf{c}(B) = w$.

First, as $\mathbf{c}(B) = w$ and the output of H is equal either to y or to n, the output $B(w)$ of B is also equal either to y or to n.

Second, when $B(w) = y$, the automaton H also outputs y. This symbol goes to the automaton A_{sw}, which switches y and n. As the result, the output of B is n. This is a contradiction, which shows that it is impossible that $B(w) = y$.

Third, when $B(w) = n$, the automaton H also outputs n. This symbol goes to the automaton A_{sw}, which switches y and n. As the result, the output of B is y. This is a contradiction, which shows that it is impossible that $B(w) = n$.

Thus, we come to a situation in which B gives no output. This is impossible. So, Theorem 7.3 is proved by contradiction.

Theorem 7.3 and Lemma 6.2 imply the following result.

Corollary 7.1. The Free Output Problem $\mathbf{F_{Oy}}$ is undecidable in **A**.

Let us consider a class **K** of algorithms/automata that satisfies postulates **PDC**, **PDS**, and **PCS,** axiom **AC**, and conditions **CA2,** $\mathbf{WLSW}_{yn}$, **CA, CS,** and **CC**.

Theorem 7.4. The Decidability Problem $\mathbf{R_D}$ is unsolvable in **K**, i.e., there is no automaton H in **K** such that for all algorithms A in **K,** and for all strings w in V^*, this automaton H can check if $A(w)$ is defined or not.

<u>Proof</u>. Let us assume that such as automaton H exists in **K**. It means that given a pair $(\mathbf{c}(A), w)$, H produces y when $A(w) = a$ for some element a from V^*, and produces n when A gives no result. Properties of the class **K** allow us to build the sequential composition $D = A_C \circ H \circ A_w$, which also belongs to **K** and in which the automaton A_w given the input y gives no output and given the input n, gives y as the output.

This composition D is presented in Figure 3.

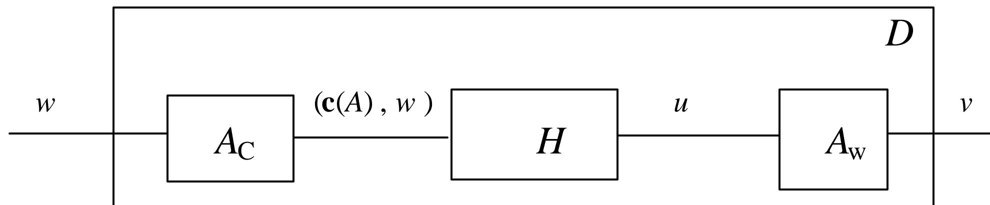


Figure 3. A hypothetical automaton D

Let us consider functioning of the automaton D when it works with the word $\mathbf{c}(D)$ $= w$.

First, as $\mathbf{c}(D) = w$, the output of H is equal either to y or to n, and the output of A_w can be only n, the output $B(w)$ of B can also be only n.

Second, when B gives no output, the automaton H outputs n. This symbol goes to the automaton A_w, which switches y and n. As the result, the output of B is y. This is a contradiction, which shows that it is impossible that B gives no output.

Third, when $B(w) = n$, the automaton H outputs y. This symbol goes to the automaton A_{sw}, which gives no output. As the result, B also gives no output. This is a contradiction, which shows that it is impossible that $B(w) = n$.

Such automaton B that neither gives no output nor gives some output cannot exist. So, Theorem 7.4 is proved by contradiction.

Let in addition the class **K** satisfies Axiom **AU**.

Corollary 7.2. The Local Decidability Problem $\mathbf{R}_{\mathbf{D}A}$ is unsolvable in **K**, i.e., there is an algorithms A in **K** such that no automaton H in **K** can check for all strings w in V^* if $A(w)$ is defined or not.

Indeed, taking a universal in **K** algorithm U as A, we see that decidability of $\mathbf{R}_{\mathbf{D}U}$ is equivalent to decidability of $\mathbf{R}_\mathbf{D}$.

Corollary 7.3. There is no finite automaton that can check for any given finite automaton if it accepts a given word, **w**.

However, Turing machines can solve this problem (cf., for example, (Hopcroft *et al*, 2001)).

Corollary 7.4. There is no recursive function that can check for any given recursive function $f(x)$ satisfiability of the condition $f(x) = n$ for a fixed number n and any given number x.

Definition 7.3. A linguistic problem **P̲** (linguistic property **P**) is called non-trivial in **K** if there is a language L generated by some automaton from **K** that has the related property **P** and there is a language L generated by some automaton from **K** that does not have the related property **P**.

Trivial linguistic properties are such properties that either there are no computable in **K** languages with such properties or all computable in **K** languages have these properties.

Condition EL. The empty language ∅ is generated by some automaton from **K**.

Remark 7.3. Condition **EL** is inconsistent with Axiom **AT**.

Let us assume that the class **K** satisfies Postulates **PDC**, **PDS**, and **PCS,** Axioms **AC** and **AU**, and Conditions **CA2, WLSW**$_{yn}$, **EL**, **CA, CS,** and **CC**.

Theorem 7.5. Any non-trivial in **K** linguistic problem **P** is unsolvable in the class **K**.

Proof. The problem **P** is related to a property **P**. Let us consider the class $\mathbf{L_P}$ of all languages that have the property **P** and assume that the empty language ∅ does not belong to $\mathbf{L_P}$. As the property **P** and the problem **P** are non-trivial in **K**, there is an automaton T from **K** such that it generates some non-empty language $L = L(T)$.

Taking some algorithm A from **K** and an arbitrary word x in the alphabet V and using conditions **G** and **GC**, we can build the automaton $B_{A,x}$ in **K** that is represented by the schema in Figure 4.

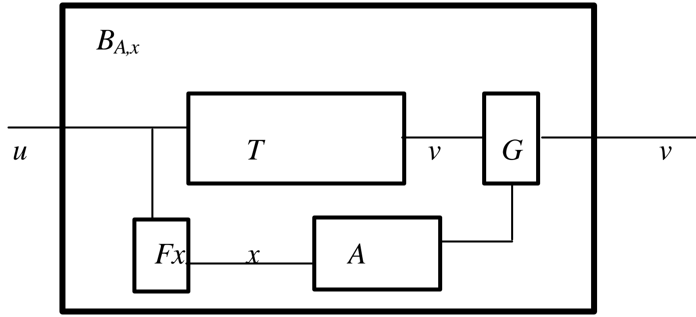


Figure 4. The structure of the machine M_Q

The automaton $B_{A,x}$ consists of four parts: T, A, Fx, and G. The automata and are described above. The automaton G performs the function of gates. Initially it is closed and does not let through anything. Any input that comes from A opens G and it gives as output its input v that comes from T. This output is also the output of $B_{A,x}$. The automaton G generates the word x, given any input, and then sends this word x to A.

This construction results in the following property of the automaton $B_{A,x}$. When A produces a result, given x as its input, the automaton $B_{A,x}$ generates the language L. When A produces no result, given x as its input, the automaton $B_{A,x}$ generates the empty language ∅. Thus, the language $L(B_{A,x})$ has the property **P** if and only if the value $A(x)$ is defined. If there is an automaton in **K** that solves the linguistic problem **P** for all automata $B_{A,x}$, then this automaton, or

its simple modification, can solve the Decidability Problem $\mathbf{R}_{\mathbf{D}A}$. However, by Corollary 7.2, this problem is unsolvable when $A = U$ is a universal algorithm in **K**. Consequently, the linguistic problem $\underline{\mathbf{P}}$ is also unsolvable.

To finish the proof, we have to consider the situation when the empty language ∅ belongs to $\mathbf{L_P}$. If the problem $\underline{\mathbf{P}}$ is solvable in **K**, then by Corollary 6.1 the problem $\underline{\neg\mathbf{P}}$ is also solvable in **K** and ∅ does not belong to the class $\mathbf{L}_{\neg\mathbf{P}}$ of all languages that do not have the property $\underline{\mathbf{P}}$. The property $\underline{\neg\mathbf{P}}$ is linguistic and nontrivial because the property $\underline{\mathbf{P}}$ is linguistic and nontrivial. However, it has been demonstrated that any problem related to a nontrivial linguistic property is unsolvable. So, both problems $\underline{\mathbf{P}}$ and $\underline{\neg\mathbf{P}}$ are unsolvable.

Theorem 7.5 is proved.

Remark 7.4. It is possible to prove Theorem 7.5 without Axiom AU, but with constructive forms of composition axioms. Constructive forms mean not only existence of all necessary compositions, but also that there is an algorithm in **K** that builds all these compositions.

Theorem 7.5 and Lemma 6.3 imply the following result.

Corollary 7.5. The Fixed Output Problem $\mathbf{R}_{\mathbf{O}y}$ is unsolvable in **K**.

Theorem 7.3 and Lemma 6.2 imply the following result.

Corollary 7.6. The Free Output Problem $\mathbf{F}_{\mathbf{O}y}$ is unsolvable in **K**.

Corollary 7.7. The problem whether an arbitrary given algorithm *A* from **K** is everywhere defined (satisfies Axiom **AT**) is unsolvable in **K**.

Corollary 7.8. The problem whether a subclass **H** of **K** satisfies Axiom **AT** is unsolvable in **K**.

Corollary 7.9 (Rice Theorem in the linguistic form, cf. (Davis, 1982; Rogers, 1987))**.** Any non-trivial linguistic property **P** of Turing machines (recursive algorithms in the sense of (Burgin, 2001)) is undecidable in the class **T** of all Turing machines (recursive algorithms).

Rice Theorem is an important result for computer science because it sets up boundaries for research in that area, as well as for software design. It informally states that only trivial properties of programs in general programming languages are algorithmically decidable.

Definition 7.4. A functional problem $\underline{\mathbf{P}}$ (functional property **P**) is called non-trivial in **K** if there is a function *f* realized by some automaton from **K** that has the related property **P** and there is a function *g* realized by some automaton from **K** that does not have the related property **P**.

Trivial functional properties are such properties that either there are no computable in **K** functions with such properties or all computable in **K** functions have these properties.

Condition EF. The function $f_{\varnothing}$, which is undefined for all elements from the domain of **K**, is realized by some automaton from **K**.

Remark 7.4. Condition **EF** is inconsistent with Axiom **AT**.

Let us assume that the class **K** satisfies Postulates **PDC**, **PDS**, and **PCS,** Axioms **AC** and **AU**, and Conditions **CA2, WLSW**$_{yn}$, **EF**, **CA, CS,** and **CC**.

Theorem 7.6. Any non-trivial in **K** functional problem **P** is unsolvable in the class **K**.

Proof. The problem **P** is related to a property **P**. Let us consider the class $\mathbf{F_P}$ of all functions g: $V^* \to V^*$ that have the property **P** and assume that the empty function $f_{\varnothing}$ does not belong to $\mathbf{F_P}$. As the property **P** and the problem **P** are non-trivial in **K**, there is an automaton T from **K** such that it realizes some non-empty functions f: $V^* \to V^*$.

Taking some algorithm A from **K** and an arbitrary word x in the alphabet V and using conditions **G** and **GC**, we can build the automaton $B_{A,x}$ in **K** that is represented by the schema in Figure 5.

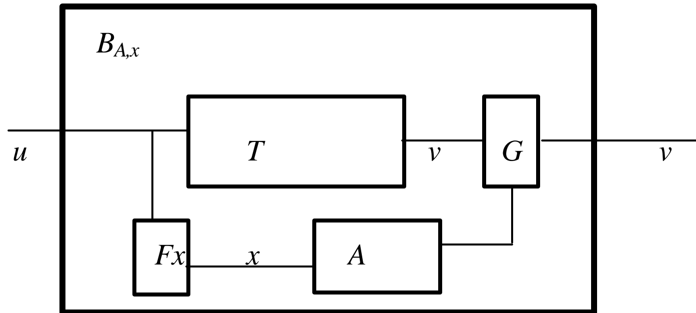


Figure 5. The structure of the machine M_Q

The automaton $B_{A,x}$ consists of four parts: T, A, Fx, and G. The automata and are described above. The automaton G performs the function of gates. Initially it is closed and does not let through anything. Any input that comes from A opens G and it gives as output its input v that comes from T. This output is also the output of $B_{A,x}$. The automaton G generates the word x, given any input, and then sends this word x to A.

This construction results in the following property of the automaton $B_{A,x}$. When A produces a result, given x as its input, the automaton $B_{A,x}$ realizes the function f. When A produces no result, given x as its input, the automaton $B_{A,x}$ realizes the empty function $f_{\varnothing}$. Thus, the function computed by the automaton $B_{A,x}$ has the property **P** if and only if the value $A(x)$ is defined. If there is an automaton in **K** that solves the functional problem **P** for all automata $B_{A,x}$, then this automaton, or its simple modification, can solve the Decidability Problem $\mathbf{R}_{\mathbf{D}A}$. However, by Corollary 7.2, this problem is unsolvable when $A = U$ is a universal algorithm in **K**. Consequently, the functional problem **P** is also unsolvable.

To finish the proof, we have to consider the situation when the empty function $f_{\varnothing}$ belongs to $\mathbf{F_P}$. If the problem **P** is solvable in **K**, then by Corollary 6.1 the problem **¬P** is also solvable in **K** and $\varnothing$ does not belong to the class $\mathbf{F}_{\mathbf{\varnothing P}}$ of all functions that do not have the property **P** . The property **¬P** is functional and nontrivial because the property **P** is functional and nontrivial. However, it has been demonstrated that any problem related to a nontrivial functional property is unsolvable. So, both problems **P** and **¬P** are unsolvable.

Theorem 7.6 is proved.

Remark 7.5. It is possible to deduce Theorem 7.5 from Theorems 7.6 and 6.1. However, a direct proof gives a better insight in the situation with unsolvability.

Remark 7.6. It is possible to prove Theorem 7.6 without Axiom AU, but with constructive forms of composition axioms.

Corollary 7.7 (Rice Theorem in the functional form (Rice, 1951)). Any non-trivial functional property **P** of Turing machines (recursive algorithms in the sense of (Burgin, 2001)) is undecidable in the class **T** of all Turing machines (recursive algorithms).

Theorem 4.1 implies that for many classes of everywhe re defined algorithms (Axioms **AT**) universal algorithms do not exist (negation of Axiom **AU**). However, there are classes of everywhere defined algorithms with universal algorithms. For instance, let us take some everywhere defined Turing machine M that works with words in the alphabet $\{1, 0\}$. If a word w in this alphabet contains 0, then it is possible to represent w in the form $w = v0u$ where the word v does not contain zeroes. This allows us to correspond to each word v in this alphabet the Turing machine T_v for which $T_v(u) = M(w)$ with $w = v0u$. This Turing machine T_v is everywhere defined and M is a universal machines for the class $\mathbf{T}_M$ of all machines T_v.

Condition OF. There is an element a in V such that the set f_a that is identically equal to a is generated by some algorithm/automaton from **A**.

Let us consider a class **A** of algorithms/automata that satisfies Postulates **PT**, **PDS**, and **PCS**, Axioms **AT**, **AC** and **AU**, and conditions **CA2**, $\mathbf{LSW}_{yn}$, **CA**, **CS**, **OF**, and **CC**.

Theorem 7.7. Any non-trivial in **A** functional problem **P** is unsolvable in the class **A**.

Condition OL. There is an element a in V such that the set $L_a = \{a\}$ is generated by some algorithm/automaton from **A**.

Theorems 7.7 and 6.1 imply the following result.

Let us consider a class **A** of algorithms/automata that satisfies Postulates **PT**, **PDS**, and **PCS**, Axioms **AT**, **AC** and **AU**, and conditions **CA2**, $\mathbf{LSW}_{yn}$, **CA**, **CS**, **OL**, and **CC**.

Theorem 7.8. Any non-trivial in **A** linguistic problem **P** is undecidable in the class **A**.

In contrast to functional and linguistic properties, some non-trivial operational properties are decidable in the class **T** of all Turing machines, while others are not.

The following properties are decidable (Sipser, 1997):

1. "The head of a Turing machine moves only to the right for a given input."
2. "The head of a Turing machine does not make turns."

The following properties are undecidable (Sipser, 1997):

1. "The head of a Turing machine returns into the first cell for a given input."
2. "The head of a Turing machine comes to the n-th cell, at least, two times for all inputs."

Some non-trivial descriptive properties are decidable in the class **T** of all Turing machines, while others are not.

The descriptive problem "Does the description d(A) of an algorithm A contain the letter "a"?" is decidable.

As results on Kolmogorov complexity and other dual complexity measures (Li and Vitanyi, 1997; Burgin, 1982) show, the following descriptive problems are undeciable:

- Is a given description d(A) of an algorithm A is minimal for all algorithms that compute the same function?
- Is a given description d(A) of an algorithm A is minimal for all algorithms that compute the word x without input?

- What is the length of the description d(A) of a minimal algorithm A that computes the word x without input?

It is interesting that although all kinds of Turing machines have the same computing power (cf., for example, (Hopckroft *et al*, 2001)), decidability of some properties of Turing machines depends on the model used. It is demonstrated by the following results.

Let us consider $n > 0$ and the property V_n:

" Given input x, the head of a Turing machine comes into the cell with number n."

Theorem 7.10. The property V_n is decidable for Turing machines with one-sided linear tape.

Proof. The property V_n is equivalent to the property U_n:

"Given input x, a Turing machine T uses only (n-1) cells for its computation."

Indeed, for Turing machines with a one-sided linear tape, the property V_n is true if and only if the property U_n is false. Thus, both of them are either decidable or undecidable.

To check the property U_n for an input x, we note that if this property is true, then T works only with words that have length less than or equal to $n - 1$. The number of such words is equal to $(p - 1)^{(n-1)}$ where p is the number of symbols in the alphabet of the Turing machine T. Let us assume that T has r states. Then T either makes $r(p - 1)^{(n-1)} + 1$ moves or halts. We can check both options, using universal Turing machine. If T makes $r(p - 1)^{(n-1)} + 1$ moves, then one of the states of T repeats with the same word on the tape according to the pigeonhole principle (cf., for example, (Hopckroft *et al*, 2001)). Such a repetition means that that T goes into an infinite cycle and does not use more than (n-1) cells.

Thus, if T uses more than (n-1) cells, i.e., the head comes to the cell with number n, it is possible to find this simulating not more than $r(p - 1)^{(n-1)} + 1$ moves of T. Consequently, the validity of U_n can be found by simulating not more than $r(p - 1)^{(n-1)} + 1$ moves of T. It means that the property U_n is decidable.

As properties V_n and U_n are equivalent for Turing machines with a one-sided linear tape, the property U_n is also decidable.

Theorem is proved.

For a slightly changed model, the same property appears to be undecidable.

Theorem 7.11. The property V_n is undecidable for Turing machines with a two-sided linear tape.

Proof. Let T be an arbitrary Turing machine with one-sided linear tape and $\mathbf{T}_{1l}$ be the class of all such machines. As usually, we assume that the one-sided tape goes from the zero cell to the right. By the initial condition, the number n in the property V_n is larger than zero. We correspond to T a Turing machine T_2 with a two-sided linear tape and the same states as T, but working in a different manner.

The input word is written in the left part of the tape of T_2, starting with the zero cell. At the beginning, T_2 works only with the left part of its tape, i.e., with cells with numbers 0, -1, -2, -3, … , starting with its head in the zero cell. The rules of T_2 consist of two parts: R_1 and R_2. The rules of the first part R_1 are obtained from the rules of the machine T by changing all left moves of the head of T to the right moves of the head of T_2 and all right moves of the head of T to the left moves of the head of T_2. As a result, T_2 imitates all moves of the machine T, with its head going to the opposite direction.

At the same time, the machine T_2 does not have final states. To each final state q of T, there is a system of corresponding rules from R_2 that have the following form

$$qa \rightarrow \mathrm{R}q$$

Here a is an arbitrary symbol from the alphabet of T or the empty symbol Λ.

As a result, when T_2 comes to some final state of T, instead of halting, the head of T_2 starts moving to the right without stopping. Consequently, given input x', the head of T_2 comes to the cell with number n if and only if T halts given input x. Here x' is obtained from the word x by its reversion, e.g., if $x = abc$, then $x' = cba$. Thus, when the problem V_n is decidable, we can build an algorithm that decides the halting problem for all Turing machines with one-sided linear tape. As the property "to halt starting with a given input" is undecidable, the property V_n is also undecidable and theorem is proved.

8. Software and hardware verification and testing

Here we show how results about abstract classes of algorithms are related to real life problems of building computers and networks, designing their software and developing information technology in general.

As an application of theoretical results obtained in previous sections, we consider problems of verification of program/device correctness and of program/device testing with the goal to eliminate bugs. At first, we reflect on software verification. As Voas and Miller (1995) write, "software verification is often the last defense against disasters caused by faulty software development. When lives and fortunes depend on software, software quality and its verification demand increased attention. As software begins to replace human decision-makers, a fundamental concern is whether a machine will be able to perform the tasks with the same level of precision as a skilled person. If not, a catastrophe may be caused by an automated system that is less reliable than a manual one."

To be able to apply theoretical results in this area, which are mostly mathematical by their nature, it is necessary to have an exact definition of software verification. Informally, verification answers the question whether a given program is correct. Although program correctness looks very simple property, experience of programmers and computer scientists demonstrates that this is a very sophisticated and complex property, which has different meanings. Programs are written to solve problems. So, a correct program p solves correctly some problem(s) P. Solving this problem is the goal of p, which is formulated to a program developer. For a long time, users simply explained what they want. Now this process is formalized, and to give an explicit and sufficiently unambiguous explanation of this goal, specifications of programs to be designed are written. There are various specification languages and different models for correctness are used (Burgin and Greibach, 2002). As a result, it is possible to define program correctness in several ways.

Definition 8.1. A program is correct if it matches the needed/demanded specification.

This is a very general and thus, not completely formalized definition of correctness because we have two undefined concepts: "to match" and "specification".

A natural way to determine a program specification in a an exact form is to correspond to a program p some function $f: X \to Y$. It is assumed that the program p has to process inputs taken from X into the outputs that belong to Y. This program

representation is reflected in the Deterministic Computation Postulate **PDC.** It gives us one of the definitions of program correctness.

Definition 8.2. The program p is *functionally correct* if it realizes/computes a function f given by specification.

Usually we consider programs written in some popular programming language, such as C^{++}, Java, ALGOL, SIMULA or PROLOG, and compare them to Turing machines.

Definition 8.3. A programming language is called Turing complete if it is functionally equivalent to the class **T** of all Turing machines.

It is, as a rule, proved that the class **K** of all programs in such languages is Turing complete. That is why these languages are called *general programming languages* as the majority of programming languages satisfies this condition.

To realize some function is a nontrivial functional property P_f. Thus, by Theorem 7.6, this property is undecidable in the class **T**. As classes **T** and **K** are functionally equivalent, we have the following result.

Theorem 8.1. It is impossible to create a software system V written in a general programming language **L** such that V verifies functional correctness of all programs written in **L**.

Let us consider some system C of logical statements of the form:

`If x has property P, then y has property Q.`

Definition 8.4. The program p is *logically correct* with respect to the system C if all statements from C are true when x belongs to the domain X of p and x belongs to the range Y of p.

It is possible to express the property P_f in the form:

If x belongs to the domain of the function f, then $p(x) = f(x)$.

As a result, Theorem 8.1 implies the following result.

Corollary 8.1. It is impossible to create a software system V written in a general programming language **L** such that V verifies logical correctness of all programs written in **L**.

This result shows that automatic program verification cannot verify correctness for all programs. In particular, testing cannot find all bugs as the following result demonstrates.

Corollary 8.2. It is impossible to create a software system *V* written in a general programming language **L** such that *V* debugs all programs written in **L**.

Indeed, according to the conventional model of algorithms, to give a result a program must halt. So, if a program does not stop, it is a result of some bug. However, if the programming language is Turing complete, it is impossible by standard methods to check whether any given program halts or not. This implies the result of Corollary 8.2.

If it is impossible to verify all programs by program verifier, it is possible to look at logical means of program verification. There is a diverse literature on this topic (cf., for example, (Loeckx *et al*, 1985) or (Colburn *et al*, 1993)). However, results from (Lewis, 2001) imply that in a general case it is also impossible for a programmer to verify correctness of all programs by logical methods.

Consequently, automatic program verification is impossible in general and programs can be too complex for such verification. Thus, computer scientists try to find ways to do software verification, at least, partially. One way is to consider not all programs but some restricted classes, for which it might be possible to build an automatic verifier. Let us simplify the verification problem, assuming that we are dealing only with such programs that always give the result. So, for conventional programs, no infinite loops are allowed and we need to check only if the program gives a correct result. For instance, we know that for all inputs the result has to be a fixed word/number. In this case, Theorem 7.3 shows that it is impossible to check this by a general algorithm. It gives us the following result.

Theorem 8.2. It is impossible to create a software system *V* written in a general programming language **L** such that *V* verifies functional correctness of all programs in **L** that always give the result.

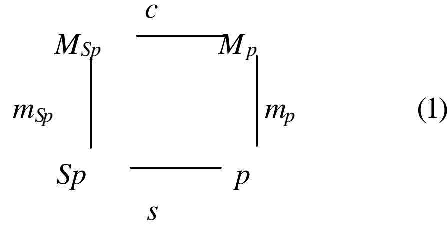
Another way is not to try to verify the program itself, but to build program models and to verify these models. To formalize this idea, we introduce even more general definition of program correctness.

Definition 8.5. A program *p* is *descriptively correct* if the program's description/model matches the needed specification.

Another way to achieve automated program verification is to make the correctness condition weaker. This is done by introduction of the concept of model program correctness.

Definition 8.6. A program p is *model correct* if the program's model matches a defined model of the needed specification.

This approach is based on the following model of program verification (Burgin and Tandon, 2003):



Here p is a program (program system), Sp is its specification, M_p is its model, and M_{Sp} is a model of the specification Sp. To be correct, the program p has to allow a correspondence c with specific properties.

Finite automata are natural models of programs. At the same time, abstract automata are used for program specification. They are especially useful for simulation program specification as automata give diverse models of simulated systems. In addition, automata are utilized for program synthesis. For example, Hune and Sandholm (2000) develop a method for synthesizing control programs. The method merges an existing control program with a control automaton. Different authors developed specification languages based on automata. Second, abstract automata are used for program understanding and documentation in description of the program behavior. Third, abstract automata are used for program testing and formal verification. A review of such approaches, including their system and methodological analysis, is given by Burgin and Greibach (2002).

Modeling programs and computing devices with abstract automata constitutes one of the main parts of computer science, which is effectively used in information technology.

Definition 8.7. An abstract automaton A is a *model* for a program (program system) P if some properties of P can be deduced from properties of A.

We consider here two types of automata models of programs: *descriptive* and *simulative.*

Definition 8.8. A model for a program P in a form of an abstract automaton A is called *simulative* if a correspondence between input, output and states of the program P and the

automaton A is specified and transitions of A represent execution of the program P for corresponding data.

Definition 8.9. A model for a program P in a form of an abstract automaton A is called *descriptive* if the automaton A is completely specified by its transition diagram (function or table) and there is a structural correspondence between input, output, and states of the program P and the automaton A.

There are different ways of building descriptive automaton models for programs. A program P can be perceived as a collection of modules with a small, algorithmic core that expresses how the modules are used to obtain a desired effect. In a sense, the core is expressed in an application specific language, with control constructs implemented by the modules. With proper separation between core and module internals one can use a distinct language for the internals, as long as the interface is meaningful seen from either side. Rather than describe *how* the "operations" of a module work, one might describe *what* they achieve. In other words, we can ascribe a system of states to each module and consider possible inputs and outputs as words in some formal language. This allows us to consider transformation of data performed by a module as the transition function of this module. In such a way, an automaton is corresponded to the program module. Combining these automata, we get an automaton that represents the whole program P.

Since a module can be seen as implementing one or more specialized language constructs, this view on program description is closely related to programming language semantics. In ascribing meaning to the expressions in a program, it is necessary to determine both how execution of the expressions will affect the underlying computer state and how the sequence of computer state changes are related to issues of human concern. To represent these issues, it is very natural to use abstract automata. This approach is represented in the Vienna definition language (Wegner, 1972; Ollongren, 1974).

For example, it is possible to retranslate logical representation of program properties, which is used in the axiomatic theory of computer programming (Floyd, 1959; Naur, 1966; Hoare, 1969), into the automata representation of programs. As Hoare writes (1969), one of the most important properties of a program is whether or not it carries out its intended function. The intended function of a program, as well as of its parts, is specified by making general assertions about the values of input and output variables. These assertions are

presented as formal expressions of the form R{P}Q where the precondition R and postcondition Q are some logical formulas and P is a program (an instruction or command). The meaning of such an expression is:

If the assertion R *is true before initiation of the program P, then the assertion* Q *is true on its termination.*

This allows us to consider logical formulas as symbols of the input and output alphabets of an automaton A that is corresponded to a program, while programs, commands, or instructions form the set of the states of A. We call this automaton the *state based automaton* of the program P and denote it by Ast(P). This is a descriptive model of P.

Another interpretation of the expression R{P}Q is given by Milner (1989):

If P is executed in a state satisfying R *and it terminates, then the terminating state will satisfy* Q.

This allows us to consider logical formulas as states of an automaton A that is corresponded to a program, while programs, commands, or instructions form symbols of the input alphabet of this automaton A. In this case, A is an accepting automaton. We call this automaton the *symbol based automaton* of the program P and denote it by Asy(P). This is also a descriptive model of P.

There are other ways to correspond automata to logical specifications of programs. For example, Hune and Sandholm (2000) utilize monadic second order logic over strings for specifying the control automata. More exactly, they use the Mona tool to translate logical specifications into automaton description.

After we build an automaton representation of both program specification and description, it is possible to use formal methods for verifying/testing their relevance and program correctness. To do this for a program P, we consider the automaton A_{sp} that is designed as a specification of P or as a model of such a specification and the automaton A_d that is built as a description of P when the program is already developed. We call A_{sp} a specification automaton of P and A_d a description automaton of P.

Definition 8.10. The program P is accepted as a correct program if a definite correspondence exists between the automata A_{sp} and A_d.

There are different kinds of natural correspondences between abstract automata. Using them, we come to different kinds of program correctness. For example, let us consider the

logical correctness, to get a proof of which is the main goal of the logical or axiomatic approach to program verification (Hoare, 1969),

According to this approach, it is assumed that a program specification is done by making definite assertions about the properties of input and output data. These assertions about a program P are presented as formal expressions of the form R{P}Q where R and Q are some logical formulas. When the program P is created, similar expressions are corresponded to the instructions that constitute this program. Then definite logical formulas that represent rules of composition are added and from all these expressions taken as axioms, the formula R{P}Q representing specification is deduced.

As we have demonstrated, it is possible to interpret each formula of the form R{C}Q as a finite automaton. We can correspond a state based automaton Ast(C) and a symbol based automaton Asy(C) to these formulas.

As a result, we relate to the logical program specification R{P}Q an automaton program specification Aw(P), which is called the whole program automaton of P.

Rules of instruction composition, both in procedural and axiomatic forms, give corresponding rules for automata composition. This allows one to construct the composite program automaton Ac(P) of P. This automaton plays the role of a program description. Usually, correctness of a program is verified by establishing some equivalence relation between the specification and description automata corresponded to the program. Definition 8.10 allows us to make this idea exact.

Definition 8.11. The program P is A-correct (FA-correct) if the automata Aw(P) and Ac(P) are equivalent (and finite).

There are different kinds of equivalence, implying different kinds of program correctness. Here we consider algebraic forms of automata equivalence. The most evident equivalence is equality.

As it is demonstrated above, to verify program correctness, it is necessary to establish a correspondence between automata. It is natural to demand that such correspondence preserves structures of these automata. To make this concept exact, we consider here only finite automata and remind the definition of the structure of a finite automaton.

According to the traditional approach, there are three forms of representation of finite automata: *analytical*, *dynamic*, and *table* representations. We begin with the *analytical form.*

In the analytical form, a finite automaton A consists of three structures, i.e., A = (L, S, δ):

- The *linguistic structure* L = (Σ, Q, Ω) 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 A;
- The *state structure* S = (Q, q_0 , F) where q_0 is an element from Q that is called the *initial* or *start state* and F is a subset of Q that is called the set of *final* (in some cases, *accepting*) states of the automaton A;
- The *action structure*, which consists of the *transition function*, or more exactly, *transition relation* of the automaton A

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

and of the *production* or *output function*, or more exactly, *production* or *output relation* of the automaton A

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

The *dynamic form* of representation of a finite automaton A is different from its analytic form only in its transition function representation. The function/relation δ is given in a form of a transition diagram.

The *table form* of representation of a finite automaton A is also different from its analytic form only in its transition function representation. The function/relation δ is given in a form of a table.

Here we consider the analytical form of representation of a finite automaton A as the most appropriate for utilizing methods of modern algebra to study properties of automata models of programs.

From the algebraic perspective, it is possible to correspond a heterogeneous algebra in the sense of Birkhoff and Lipson (1970) and Mathienssen (1978) to any deterministic finite automaton A.

Definition 8.12. A *heterogeneous algebra* U is a set U (the support of U) with a system of operations Σ in which elements of U form an indexed system $U = \{A_i ; i \in I\}$ of sets and each operation is a mapping having the form $f: A_{i1} \times A_{i2} \times \ldots \times A_{ik} \to A_i$.

Other examples of heterogeneous universal algebras are modules, polygons, i.e., sets on which monoids act, polyadic or Halmos algebras (Halmos, 1962), nonhomogeniuos polyadic algebras (Leblanc, 1962), relational algebras (Beniaminov, 1979), and state machines.

Heterogeneous universal algebras were studied by several authors under different names. To mention only some of them, it is necessary to name algebras with a scheme of operators introduced by Higgins (1963; 1973), multibase universal algebras (Glushkov, et al, 1974; Shaposhnikov, 1999; Karpunin and Shaposhnikov, 2000), and many-sorted algebras studied by Plotkin (1991). The term "heterogeneous algebras" is used more often than other related terms. Heterogeneous (multibase or many-sorted) algebras represent the next level of the development of algebra. Namely, in ordinary (or homogeneous) universal algebras operations are defined on a set, while in heterogeneous algebras operations are defined on a named set (Burgin, 1990). This makes possible to develop more adequate models for many processes and systems. For example, heterogeneous algebras are extensively used for mathematical modeling information processing by computers. Such models as abstract automata and abstract states machines or evolving algebras become more and more widespread in computer science. In addition, relational algebras are extensively used for modeling relational databases (Beniaminov, 1979; Plotkin, 1991).

It is necessary to remark that transition from a set to a named set as a basic structure goes on not only for algebra but for other fields (cf. (Burgin, 1990)). Thus, in many cases fibers, which are special cases of topological named sets, replace topological spaces in topology. Multivalued and multi-sorted logics are becoming more and more popular in logic. Manifolds are used instead of Euclidean spaces in mathematical analysis. Modern combinatorics is built on multisets, which are special cases of named sets.

Finite automata and state machines are directly connected to heterogeneous algebras. A specific heterogeneous algebra Al(A) is corresponded to a deterministic finite automaton A in the following way. This algebra Al(A) has the support { Σ, Q, Ω }, two binary operations $\delta: \Sigma \times Q \to Q$, and $\sigma: \Sigma \times Q \to \Omega$, and several unary operations $\sigma_0, \sigma_1, \dots, \sigma_k$ on the set Q: $\sigma_0 = q_0, \sigma_1 = q_1, \dots, \sigma_k = q_k$ with $q_1, \dots, q_k \in F$.

In addition to the algebra Al(A), another heterogeneous algebra EAl(A) is also corresponded to the automaton A. This algebra EAl(A) has the support { Σ^*, Q, Ω^* } where Σ^* and Ω^* are the sets of all finite words in the alphabets Σ and Ω, correspondingly. Operations of EAl(A) consist of two binary operations $\delta^*: \Sigma^* \times Q \to Q$, and $\sigma^*: \Sigma^* \times Q \to \Omega^*$, and several unary operations $\sigma_0, \sigma_1, \dots, \sigma_k$ on the set Q where $\sigma_0 = q_0, \sigma_1 = q_1, \dots, \sigma_k = q_k$ with $q_1, \dots, q_k \in F$.

Usually, if an automaton A represents a program, then a state of A is a set of states of the program variables. Inputs for programs are also collections of data. Consequently, we come to the conclusion that Al(A) and EAl(A) have operations with arity higher than two.

The same constructions allow one to correspond to a nondeterministic finite automaton B an algebraic system in the sense of Malcev (1970). This system is not a (universal) algebra in a general case. Here for simplicity, we consider only deterministic automata. However, it is possible to extend the main constructions and results to the nondeterministic case.

Definition 8.13. The program P is *strictly* A-*correct* (FA-*correct*) if the automata Aw(P) and Ac(P) are (finite and) equal as heterogeneous algebras (algebraic systems).

The assumption that automata Aw(P) and Ac(P) are finite allows us to solve the problem of program verification because it is possible to check if two finite heterogeneous algebras are equal.

Theorem 8.3. The property of program strict FA-correctness is decidable.

Informally, it means that in contrast to results of Theorems 8.1 and 8.2 it is possible to build a software system such that it verifies strict program correctness for all programs.

Applying the algebraic approach, we use another popular kind of equivalence of algebraic systems, which is called isomorphism (Cohn, 1965; Kurosh, 1974).

Definition 8.14. The program P *is isomorphically* A-*correct* (FA-*correct*) if (the automata Aw(P) and Ac(P) are finite and) the algebras Al(Aw(P)) and Al(Ac(P)) of the automata Aw(P) and Ac(P) are isomorphic as heterogeneous algebras (algebraic systems).

It is possible to demand validity of a weaker condition in a definition of correctness.

Let us assume that all states of the automata Aw(P) and Ac(P) representing programs from a class **P** are distinguishable.

Theorem 8.4. The property of program isomorphic FA-correctness is decidable for the class **P**.

Indeed, if we take two finite heterogeneous algebras, there is only a finite number of mapping between them. Utilizing properties of universal algebras (cf. Kurosh, 1974), it is possible to build an algorithm that checks if any of these mapping is an isomorphism. In such a way, we can algorithmically find whether two finite heterogeneous algebras are isomorphic or not.

Corollary 8.3. If a program is equivalent as algorithm to a finite automaton, then its correctness is decidable.

Having different types of program correctness, it is possible to compare them by their verification power. This idea is formalized in the following definition.

Let us consider two types X and Z of program correctness.

Definition 8.15. X-correctness is stronger than Z-correctness if for any program P, X-correctness implies Z-correctness.

Definitions 8.13, 8.14, and15 imply the following result.

Proposition 8.1. The strict A-correctness (FA-correctness) is stronger than isomorphic A-correctness (FA-correctness).

Remark 8.1. However, if we restrict the class of possible verificators only to programs that are equivalent to finite automata, then verification becomes once more impossible. In other words, the verifying program has to be more powerful than a finite automaton.

We can apply theoretical results on problem solvability not only to software, but also to hardware, utilizing general results from computer science. A general assumption in computer science is that any general purpose computer is equivalent to a universal Turing machine (e.g., (Hopcroft *et al*, 2001)). This makes it possible to apply Theorem 8.1 to hardware and get the following statement.

Theorem 8.7. It is impossible to create a software system *V* written in a general programming language **L** such that *V* verifies for all programs whether a given program computes on a general purpose computer a given partial recursive function.

Taking some system of operations, it is possible to consider all possible computers with these operations. In this context, it is impossible to find if an arbitrary computing device is equivalent by its functions to a general purpose computer.

Theorem 8.8. It is impossible to create a software system *V* written in a general programming language **L** such that *V* verifies for all possible computers whether a given computer is a general purpose computer.

Definition 8.16. (Wikipedia) A programming language is called *Turing-complete* if it is potentially equivalent in power to the class of all Turing machines.

In a similar way, taking some system of operators, it is possible to consider all possible programming languages with these operators.

Theorem 8.9. It is impossible to create a software system *V* written in a general programming language **L** such that *V* verifies all possible programming languages whether a given programming language **C** is a Turing complete.

9. Conclusion

We explored some basic properties of the theory of algorithms and computation. In doing so, we developed an axiomatic setting for this theory. In contrast to the traditional global axiomatization prevalent in mathematics and local axiomatization developed in computer science for theory of programs, we utilize multiglobal axiomatization aimed at study of diverse classes of algorithms and automata. This axiomatization is based on postulates, axioms, and formal conditions. Postulates represent the most basic properties of algorithmic classes. Axioms describe various essential peculiarities, while conditions reflect specific features used for derivation more important and deep traits of algorithms and computation.

Results comparing different modes of functioning and power of classes of algorithms, programs and computers are proved under very general axioms or conditions. This makes

possible to apply these results to a vast variety of types and kinds of algorithms and their models. Such models may be structurally distinct like Turing machines and partial recursive functions. They may be defined by some restrictions inside the same class of models, e.g., polynomial time Turing machines, polynomial space Turing machines, and logarithmic time Turing machines. In its turn, comparing different models allows one to obtain relations between corresponding types of computers and software systems. For example, deterministic Turing machines model conventional computers, while nondeterministic Turing machines model quantum computers. As a result, properties of deterministic Turing machines reflect properties of conventional computers, while properties of nondeterministic Turing machines reflect properties of quantum computers.

In addition, axiomatic approach allows one to obtain automatically many classical results of the conventional computability, which are considered in many textbooks and monographs (cf., for example, Manna, 1974; Davis and Weyuker, 1983; Hopcroft *et al*, 2001; Rogers, 1987).

An important peculiarity of the axiomatic theory of algorithms and computation is that now its development has become more urgent than in the first period of the algorithm theory development. The main feature of the first period of the algorithm theory development was belief of the vast majority of computer scientists and mathematicians in the, so-called, Church-Turing Thesis. In one of its forms, the Thesis claims that *any problem that can be solved by an algorithm can be solved by some Turing machine and any algorithmic computation can be done by some Turing machine*. Most of what we understand about algorithms and their limitations is based on our understanding of Turing machines and other conventional models of algorithms. The Church-Turing Thesis claims that Turing machines give a full understanding of computer possibilities. However, in spite of this Thesis, conventional models of algorithms, such as Turing machines, do not give a relevant representation of a notion of algorithm. That is why an extension of conventional models has been developed. This extension is based on the following observation. The main stereotype for algorithms states that an algorithm has to stop when it gives a result. This is the main problem that hinders the development of computers. When we understand that computation can go on but we can get what we need, then we go beyond our prejudices and immensely extend computing power.

The new models are called super-recursive algorithms. They provide for a much more computing power. This is proved mathematically (Burgin, 1988). At the same time, they give more adequate models for modern computers and Internet. These models change the essence of computation going beyond the Church-Turing Thesis (here we give a sketch of a proof for this statement) and form, consequently, a base for a new computational paradigm, or metaphor as says Winograd. Problems that are unsolvable for conventional algorithmic devices become tractable for super-recursive algorithms. The new paradigm gives a better insight into the functioning of the mind opening new perspectives for artificial intelligence.

As a result, an absolute class of algorithms disappeared and now computer scientists have to deal with a huge diversity of different classes, models, and kinds of algorithms and automata. Thus, methods that study different classes (such as multiglobal axiomatization) are more efficient than any technique (such as constructive approach or global axiomatization) that studies only one class of algorithms, even if this class is so useful and popular as the class of all Turing machines.

Being applied to program correctness, theoretical results of this paper show that, on one hand, it is impossible to find an algorithm or build a program that verifies correctness of all programs in some Turing-complete programming language. On the other hand, when a program and its specification are represented by appropriate models, such as, for example, finite automata, then it is possible to verify its correctness.

It is also necessary to remark that programs realizing inductive computations can do much more than programs realizing recursive computations (Burgin, 2001; 2003). As a consequence, it is possible to build programs that work in the inductive mode and verify correctness of much larger classes of programs than recursive verifiers. In addition, inductive verifiers can check correctness in a stronger sense.

To conclude, we formulate some open problems of the axiomatic theory of algorithms and computation.

Problem 1. What undecidability results from Section 7 is it possible to prove, utilizing the Local Universality Axiom **LAU** instead of the Universality Axiom **AU**?

Local Universality Axiom LAU. For any finite number of algorithms/automata A_1, A_2, … , A_n from **K** there is some coding **c : K ®** V^+, there is a subclass **H** of **K** such that all A_1, A_2, … , A_n belong to **H** and **H** satisfies the Universality Axiom **AU**.

Problem 2. What other classes of algorithms/automata, besides classes of all recursive algorithms, such as Turing machines, satisfy the Universality Axiom **AU**?

Problem 3. Characterize all those operational (performance, descriptive) properties that are decidable in the class **T** of all Turing machines.

Problem 4. What axiomatic classes of algorithms/automata have the fixed point property (cf., (Rogers, 1987))?

Problem 5. Study computable and decidable languages (sets) in axiomatic classes of algorithms/automata.

Problem 6. Study complexity of algorithms and computations in axiomatic classes of algorithms/automata.

Problem 7. Study dual complexity measures (Burgin, 1982) of algorithms and computations in axiomatic classes of algorithms/automata.

References

1. Abramson, F.G. (1971) Effective Computation over the Real Numbers, *12th Annual Symposium on Switching and Automata Theory*, Northridge, Calif.: Institute of Electrical and Electronics Engineers
2. Adamék, J. (1975) Automata and categories, finiteness contra minimality, *Lecture Notes in Computer Science*, 32, Springer Verlag, Berlin/Heidelberg/ New York, pp. 160-166
3. Adámek, J. and Trnková, V. (1990) *Automata and Algebras in Categories*, Kluwer Academic PublishersBarrett, E., ed. *Text, Context and Hypertext: Writing with and for the Computer*, Cambridge: MIT Press
4. Adleman, L., and Blum, M. (1991) Inductive Inference and Unsolvability, *Journal of Symbolic Logic*, v. 56, No. 3, pp. 891-900
5. Alt, F.L. (1997) End-Running Human Intelligence, in "*Beyond Calculation: The Next Fifty Years of Computing*", Copernicus, pp. 127-134
6. Barrett, E., ed. *Text, Context and Hypertext: Writing with and for the Computer*, MIT Press, Cambridge, 1988

7. Beniaminov, E.M. (1988) An Algebraic Approach to Models of Databases, *Semiotics and Informatics*, v. 14, 1979, pp. 44-80

8. Birkhoff, G. and Lipson, J.D. Heterogeneous algebras, *J. Combinatorial Theory*, *8*, 1970, 115-133.

9. Black, R. (2000) Proving Church's Thesis, *Philos. Math.,* v. 8, No. 3, pp. 244-258

10. Blum, L., Cucker, F., Shub, M., and Smale, S. *Complexity of Real Computation*, Springer, New York, 1998

11. Blum M. (1967) On the Size of Machines, *Information and Control*, v. 11, pp. 257-265

12. Blum M. (1967a) A Machine -independent Theory of Complexity of Recursive Functions, *Journal of the ACM*, v. 14, No.2, pp. 322-336

13. Büchi, J.R. (1960) Weak second order arithmetic and finite automata, Z. Math. Logic and Grudl. Math., v. 6, No. 1, pp. 66-92

14. Budach, L. and Hoehnke, H.-J. (1975) *Automata und Funktoren* , Akademic Verlag, Berlin

15. Burgin, M. S. (1982) Generalized Kolmogorov complexity and duality in theory of computations , *Notices of the Academy of Sciences of the USSR*, v. 264, No. 2, pp. 19-23 (translated from Russian, v. 25, No. 3)

16. Burgin, M. (1982a) Products of operators of multidimensional structured model of systems, *Mathematical Social Sciences*, No.2, pp. 335-343

17. Burgin, M. S. (1983) Inductive Turing Machines, *Notices of the Academy of Sciences of the USSR*, v. 270, No. 6, pp. 1289-1293 (translated from Russian, v. 27, No. 3)

18. M.Burgin, Multiple computations and Kolmogorov complexity for such processes, *Notices of the Academy of Sciences of the USSR*, 1983, v. 269, No. 4 , pp. 793-797 (transla ted from Russian, v. 27, No. 2)

19. M.Burgin, Systems and properties, *Abstracts presented to the American Mathematical Society*, 1984, v.5, No.6

20. Burgin, M. S. (1985) Algorithms and Algorithmic Problems, *Programming*, No. 4, pp. 3-14 (*Programming and Computer Software*, v. 11, No. 4) (translated from Russian)

21. Burgin M.S. (1990) Theory of Named Sets as a Foundational Basis for Mathematics, in "*Structures in Mathematical Theories*", San Sebastian, pp. 417-420

22. Burgin M.S. *Fundamental Structures of Knowledge and Information*, Ukrainian Academy of Information Sciences, Kiev, 1997 (in Russian)

23. Burgin, M. *On the Essence and Nature of Mathematics*, Ukrainian Academy of Information Sciences, Kiev, 1998 (in Russian)

24. Burgin, M. (2001) How We Know What Technology Can Do, *Communications of the ACM*, v. 44, No. 11, pp. 82-88

25. Burgin, M. (2003) Nonlinear Phenomena in Spaces of Algorithms, *International Journal of Computer Mathematics*, v. 80, No. 12, pp. 1449-1476

26. Burgin, M. and Greibach, S. (2002) Abstract Automata as a Tool for Developing Simulation Software, in "*Proceedings of the Business and Industry Simulation Symposium*," Society for Modeling and Simulation International, San Diego, California, pp. 176-180

27. Burgin, M. and Karasik, A. (1975) A study of an abstract model of computers, *Programming and Computer Software*, No. 1, pp. 72-82

28. Burgin, M. and Kuznetsov, V. (1994) *Introduction to the Modern Exact Methodology of Science*, International Science Foundation, Moscow (in Russian)

29. Burgin, M., Liu, D., and Karplus, W. (2001) The Problem of Time Scales in Computer Visualization, in "*Computational Science*", Lecture Notes in Computer Science, v. 2074, part II, pp.728-737

30. Burgin, M., Liu, D., and Karplus, W. (2001a) *Visualization in Human-Computer Interaction*, UCLA, Computer Science Department, Report CSD – 010010, Los Angeles, July, 2001, 108 p.

31. Burgin, M. and Tandon A. (2003) Software Verification in Algebraic Setting, in Proceedings of the 7th *IASTED International Conference on Software Engineering and Applications*, Marina Del Rey, California, pp. 499-504

32. Burton, D.M. *The History of Mathematics*, The McGrow Hill Co., New York, 1997

33. Buss, S.L., Kechris, A.S., Pillay, A, and Shore, R.A. (2001) The prospects for Mathematical Logic in the Twenty First Century, *Bulletin of Symbolic Logic*, v. &, No. 2, pp.169-196

34. Church, A. (1957) Application of Recursive Arithmetic to the Problem of Circuit Synthesis, in "*Summaries of Talks presented at the Summer Institute of Symbolic Logic at Cornell University*," 1, pp. 3-50

35. Cleland, C.E. (2001) Recipes, Algorithms, and Programs, *Minds and Machines*, v. 11, pp. 219-237

36. Cohn, P.M. *Universal algebra*, New York/ Evanston/London, Harper & Row, 1965

37. Colburn, T.R., Fetzer, J.H., and Rankin, T.L. (Eds.) *Program Verification*, Kluwer Academic Publishers, Dordrecht, 1993

38. Davis, M. Computability and Unsolvability. New York: Dover, 1982.

39. Feferman, S. (1993) Working foundations - `91, in *Bridging the Gap: Philosophy, Mathematics and Physics,* Boston Studies in the Philos. of Science, Kluwer, Dordrecht, vol. 140, pp. 99-124

40. Feferman, S. (1993a) What rests on what? The proof-theoretic analysis of mathematics, in *Philosophy of Mathematics*, Proceedings of the 15th International

Wittgenstein Symposium, Verlag Hölder-Pichler-Tempsky, Vienna, Part I, pp. 147-171

41. Fischer, P.C. (1965) Multi-tape and infinite-state automata, *Communications of the ACM*, v. 8, No. 12, pp. 799-805
42. Floyd, R.W. (1967) Assigning meanings to programs, *Mathematical Aspects of Computer Science, 19th Symp. of Appl. Math., AMS*, Providence, Rhode Island, 19-32.
43. Fraenkel, A.A. and Bar-Hillel, Y. *Foundations of Set Theory*, North Holland P.C., Amsterdam, 1958
44. Friedman H., and Hirst, J. (1990) Weak comparability of well orderings and reverse mathematics, Annals of Pure and Applied Logic, v. 47, pp. 11-29.
45. Giusto, M. and Simpson, S. G. (2000) Located Sets and Reverse Mathematics. The Journal of Symbolic Logic, v. 65, No. 3, pp. 1451-1480
46. Glushkov, V.M., Zeitlin, G.E., and Yushchenko, E.L. *Algebra, Languages, Programming*, Kiev, Naukova Dumka, 1974 (in Russian)
47. Gödel, K (1931) Über formal unentscheidbare Sätze der Principia Mathematics und verwandter System I, Monatshefte für Mathematik und Physik, b. 38, s.173-198
48. Goldin, D. and Wegner, P. *Persistent Turing Machines*, Brown University Technical Report, 1988
49. Halmos, P.R. *Algebraic Logic*, New York, 1962
50. Hamkins, J.D., and Lewis, A. (2000) Infinite time Turing machines, Journal of Symbolic Logic, v. 65, No. 3, pp. 567-604
51. Higgins, P.J. (1963) Algebras with a scheme of operators, *Math. Nachrichten*, v. 27, No. 1-2, pp. 115-132
52. Higgins, P.J. *Grupoids and categories*, North-Holland, 1973
53. Hoare, C.A.R. (1969) An Axiomatic Basis for Computer Programming, *Communications of ACM*, v. 12, pp. 576-580, 583
54. Hopcroft, J.E., Motwani, R., and Ullman, J.D. *Introduction to Automata Theory, Languages, and Computation*, Addison Wesley, Boston/San Francisco/New York, 2001
55. Humby, E. *Programs from Decision Tables*, MacDonald, London, 1973
56. Hune, T.S. and Sandholm, A.B. (2000) Using Automata in Control Synthesis - A Case Study, in *Fundamental Approaches to Software Engineering*, LNCS 1783, pp. 349-362
57. Karpunin, G. A. and Shaposhnikov, I. G. (2000) Crossed homomorphisms of finite multibase universal algebras with binary operations, *Discrete Math. Appl.* v. 10, no. 2, pp. 183-202

58. Kleene, S. (1956) Representation of events in nerve nets, *Automata Studies*, Princeton University Press, Princeton, N.J. pp. 3-41

59. Knuth, D. *The Art of Computer Programming*, v.2: *Seminumerical Algorithms*, Addison-Wesley, 1981

60. Kogge, P. *The Architecture of Pipeline Computers*, McGraw Hill, 1981

61. Kurosh, A.G. *General algebra*, Moscow, Nauka Press, 1974 (in Russian)

62. Kolmogorov, A.N. (1953) On the Concept of Algorithm, *Russian Mathematical Surveys*, v. 8, No. 4, pp. 175-176

63. Krinitsky, N.A. *Algorithms around us*, Moscow, Nauka, 1977 (in Russian)

64. Landow, G. *Hypertext: The Convergence of Contemporary Critical Theory and Technology*, Baltimare: Johns Hopkins University Press, 1992

65. Leblanc, L. (1962) Nonhomogeniuos Polyadic Algebras, *Proc. American Mathematical Society*, v. 13, No. 1, pp. 59-65

66. Lewis, J.P. (2001) Limits to Software Estimation, *Software Engineering Notes*, v. 26, No. 4, pp.54-59

67. Li, M., and Vitanyi, P. *An Introduction to Kolmogorov Complexity and its Applications*, Springer-Verlag, New York, 1997

68. Loeckx, J., Sieber, K. and Stansifer, R.D. *The foundations of program verification*, John Wiley & Sons, Inc. New York, 1985

69. Malcev, A.I. *Algebraic systems*, Moscow, Nauka, 1970 (in Russian)

70. Manna, Z. and Pnueli, A. *Specification and Verification of Concurrent Programs by "-Automata*, Computer Science Department, Stanford University, Computer Science Department, Weizmann Institute of Science, 1986

71. Mathienssen, G. (1978) A heterogeneous algebraic approach to some problems in automata theory, many-valued logics and other topics, *Contr. to General Algebra*, Proc. Klagenfurt Conf.

72. Mealy, G.H. (1953) A method for synthesizing sequential circuits, *Bell System Techn. J.*, v. 34, pp. 1045-1079

73. Milner, M. *Communication and concurrency*, Prentice Hall, New York/London/Toronto, 1989

74. Moore, E.F. (1956) Gedanken-experiments on sequential machines, in *Automata Studies*, Princeton University Press, Princeton, N.J. pp. 129-153

75. Moore, C. (1996) Recursion Theory on the Reals and Continuous-time Computation: Real numbers and computers, *Theoretical Computer Science,* 162, No. 1, pp. 23-44

76. Moschovakis, Ya. (2001) What is an Algorithm?, in "*Mathematics Unlimited*: 2001 *and Beyond*", Springer, New York

77. Ollongren, A. *Definition of programming languages by interpreting automata*, Academic Press, New York/London, 1974

78. Naur, P. (1966) Proofs of algorithms by general snapshots, *BIT*, v. 6, pp. 310-316.
79. von Neumann, J. (1951) The general and logical theory of automata. in "*Cerebral Mechanisms in Behavior*," The Hixon Symposium, Willey , New York, pp. 1-31
80. Nielsen, J. *Hypertext and hypermedia*, New York: Academic Press, 1990
81. Plotkin B.I. *Universal Algebra, Algebraic Logic, and Databases*, Moscow, Nauka, 1991 (in Russian)
82. Rabin, M.O. (1969) Decidability of Second-order Theories and Automata on Infinite Trees, Transactions of the AMS, v. 141, pp. 1-35
83. Rabin, M.O., and Scott, D. (1959) Finite Automata and Their Decision Problems, *IBM Journal of Research and Development*, v. 3, pp. 114-125
84. Rice, H.G. (1951) Recursive Real Numbers, *Proceedings of the AMS*, v. 5, pp. 784-791
85. Rice, H. G. (1953) Classes of Recursively Enumerable Sets and Their Decision Problems, *Trans. Amer. Math. Soc.*, v. 74, pp. 358-366
86. Rogers, H. *Theory of Recursive Functions and Effective Comp utability*, MIT Press, Cambridge, Massachusetts, 1987
87. S.J. Russel and P. Norvig *Artificial Intelligence: A Modern Approach* , Prentice-Hall, Englewood Cliffs, N.J., 1995
88. Shannon, C. (1941) Mathematical Theory of the Differential Analyzer, *J. Math. Physics*, MIT, v. 20, pp. 337-354
89. Shaposhnikov, I. G. (1999) Congruences of finite multibase universal algebras, *Discrete Math. Appl.* **9,** no. 4, pp. 403-418
90. Sipser, M. *Introduction to the Theory of Computation*, PWS Publishing C., Boston, 1997
91. Trahtenbrot, B.A. and Barzdin, J.M. *Finite Automata: Behavior and Synthesis*, Moscow, Nauka, 1970 (in Russian)
92. Turing, A. (1936) On Computable Numbers with an Application to the Entscheidungs -problem, *Proc. Lond. Math. Soc.*, Ser.2, v. 42, pp. 230-265
93. Van Leeuwen, J. and Wiedermann, J. (1985) Array Processing Machines, in *Fundametals of Computation Theory,* Lecture Notes in Computer Science, 199, Springer-Verlag, New York/Berlin, pp. 99-113
94. Van Leeuwen, J. and Wiedermann J. *A computational model of interaction*, Techn. Rep. Dept. of Computer Science, Utrecht University, Utrecht, 2000
95. Van Leeuwen, J. and Wiedermann, J. (2000b) On the Power of Interactive Computing, *Proceedings of the IFIP Theoretical Computer Science* 2000, pp. 619-623
96. Vardi, M.Y. and Wolper P. (1986) An automata-theoretic approach to automatic program verification, in *Proceedings of the 1st Annual Symposium on Logic in Computer Science*, pp. 322-331

97. Vardi, M.Y. and Wolper, P. (1994) Reasoning about Infinite Computations, *Information and Computation*, v. 115, No.1, pp. 1—37

98. Voas, J.M. and Miller, K.W. (1995) Software Testability: The New Verification, *IEEE Software,* v. 12, No. 3, pp. 17-28

99. Wegner, P. (1972) The Vienna Definition Language, *ACM Computing Surveys*, v. 4, No. 1, pp. 5-63.

100. Wegner, P. (1998) Interactive Foundations of Computing. Theoretical aspects of coordination languages. *Theoretical Computer Science*, v. 192, no. 2, pp. 315-351

101. Wikipedia, http://en.wikipedia.org/wiki/Turing-complete