EasyChair Preprint

№ 9222

Information Theoretical Principles of Software Development

Mark Burgin and Rao Mikkilineni

EasyChair preprints are intended for rapid dissemination of research results and are integrated with the rest of EasyChair.

November 2, 2022

Information Theoretical Principles of Software Development

Mark Burgin[*], Rao Mikkilineni[†]

Abstract

Contemporary software, which started from a simple observation by Alan Turing that a man in the process of computing numerical functions can be replaced by a machine that is capable of only a finite number of simple operations, has become a powerful means for advanced machines, automating business processes, and mimicking behaviors learned through processing big data. Software is all pervasive and touches us in some form or another by facilitating communication, collaboration, and commerce on a global scale. However, on the flip side, it also threatens our privacy and data security. Despite many layers of complex systems and processes attempting to protect our software systems, they are still vulnerable to hackers and fraudsters exploiting global access through the Internet.

In this paper, we argue that the increased complexity as well as the resulting instabilities and inefficiencies are a result of foundational issues of the computing model upon which current software systems are built. We use the general theory of information to suggest ways to improve the current state of the art of software development, deployment, and operation through the infusion of self-monitoring, self-regulation, and self-protection processes at both the component and system levels. A new computing model using super-symbolic computing provides an autopoietic, and cognitive process overlay over the current symbolic and sub-symbolic computing structures without having to change them. This software system is analogous to biological systems using the mammalian neocortex to manage the knowledge obtained from various reptilian cortical columns using embedded, embodied, extended and enactive cognition.

Introduction

With a humble beginning of being a set of data, instructions, and programs used to operate computers and execute specific tasks, the software has evolved to become the human knowledge about a specific domain captured as digital workflows in executable form. Beginning from encapsulating knowledge about the process of computing with numbers in executable form (using the storied program control implementation of the Turing Machine), the software has helped us to execute operations not only on numbers, but also on multimedia and other advanced structures, and enables us to communicate, collaborate, and conduct commerce on a large scale by exchanging information. It is not an exaggeration to say that software has become the lifeblood of contemporary civilization and impacts our daily lives on conscious levels as well as on the levels unknown to us. Software controls many of the systems and activities that impact our everyday lives. While software has brought many conveniences into our daily lives both personal, and professional, we are also highly vulnerable to many of the shortcomings resulting from our software dependency. Our privacy, personal and business data security, and safety from disruption of our lives by hackers, and fraudsters are at high risk. Despite layers of solutions to improve the safety, security, stability, and survival of software systems, the complexity, and cost of

[*] University of California, Los Angeles, California, USA
[†] Golden Gate University, San Francisco, California, USA

these systems have only increased with no significant improvement in addressing the risk and its mitigation (Tisdale, 2015; Burgin, Mikkilineni, 2022).

In this paper, we argue that the existing shortcomings are foundational in nature originating from the current practice of using symbolic and sub-symbolic computing software systems based on the stored program implementation of the Turing Machine. As the requirements of scale, resiliency, and efficiency become more demanding, current computing paradigms based on the Turing Machine model of computation that represents programs, and data as sequences of symbols with various operations defined to execute algorithm-based task definitions, reach their limits to meet the new requirements. The result is higher complexity, cost, and unreliability of operational stability, safety, security, and survival of systems that depend on software (Razian, et al, 2022).

We take the cue from biological systems (Mikkilineni, 2022) that have devised the ultimate software system with a high degree of scale, resiliency, and efficiency using the genome as a model for the specification and execution of the system requirements. Genome is a good example of working software architecture that integrates the processes of replication, and metabolism to execute the specification of an autopoietic and cognitive system with self-regulation that maintains stability and achieves the goal of managing the interactions of various components within the system and its interactions with the external world consisting of material structures. It uses metabolism to exploit matter and energy transformations to create symbolic and sub-symbolic information processing structures - the networks of genes (symbolic) and neurons (sub-symbolic). Sentience, resilience, and intelligence (local, clustered, and system-wide) are outcomes of cognitive and autopoietic behaviors implemented as super-symbolic structures. Autopoiesis refers to the behavior of a system that replicates itself and maintains identity and stability while facing fluctuations caused by external influences. Cognitive behaviors model the system's state, sense internal and external changes, analyze the observations, predict, and take action to mitigate any risk to its functional fulfillment.

In addition, we use the General Theory of information (GTI) (Burgin, 2010; Burgin, 2016; Burgin, Mikkilineni, 2021), which does not only provide a model to explain how the genome specifies and executes replication and metabolism to exhibit autopoietic and cognitive behaviors using symbolic, sub-symbolic, and super-symbolic computing, but also provides a schema, and associated operations to model and execute digital software systems with a super-symbolic computing structure to execute both autopoietic and cognitive behaviors. The super-symbolic computing structure is an overlay over current digital symbolic, and sub-symbolic structures very similar to the neocortex in the brain integrating the knowledge derived from information gathered from symbolic, and sub-symbolic computing structures.

In section 2, we define software as operational process knowledge and show both the genome and the digital computing systems based on the Turing Machine computing model as examples of executable operational process knowledge. In Section 3, we present theoretical models of algorithms and computation used as the base for software system development. By their nature, these models are implicitly or explicitly based on the GTI. This provides various tools for software construction such as structural machines, knowledge structures, generalized oracles as operational agents, and super-symbolic computing allowing infusion of autopoietic and cognitive behaviors into digital systems. In Section 4, we discuss the operational aspects of the GTI and its utilization in software specification and execution. In Section 5, we discuss the relationship between data, information, and the conversion of information into operational knowledge. In Section 6, we present a new approach to applying the

principles of the GTI to building the next generation of software systems with improved resilience and efficiency at scale. The new approach brings the super-symbolic computing structures integrating current generation symbolic, and sub-symbolic computing structures. As mentioned earlier, this is analogous to how biological systems evolved the neocortex structure integrating the reptilian cortical column-based 5E (embedded, embodied, extended, elevated, and enacted) cognitive machines. Finally, we conclude by summarizing the application of the GTI-based approach to software specification and execution of domain-specific knowledge and suggest future directions for improving the current state-of-the-art information technologies.

Software as Operational Knowledge

Digital software is a set of instructions (a program) represented in the form of sequences of symbols 0 and 1, used to operate computers and execute specific tasks using data, also represented in the form of sequences of symbols 0 and 1. Data includes any observables represented by a sequence of symbols in the form of labels. The observables are either physical entities in the material world or mentally conceived entities in the mental worlds of biological systems. Examples are cat, dog, point, triangle, etc.

It is the opposite of hardware, which describes the physical aspects of a computer that provide the required processing structures for executing the software (providing the metabolism that uses energy to perform specified tasks using material structures) with a processing unit, and memory. Figure 1 shows the stored program implementation of the Turing machine showing both software and hardware. A computational operator defines the operations on the data converting the input to the output.

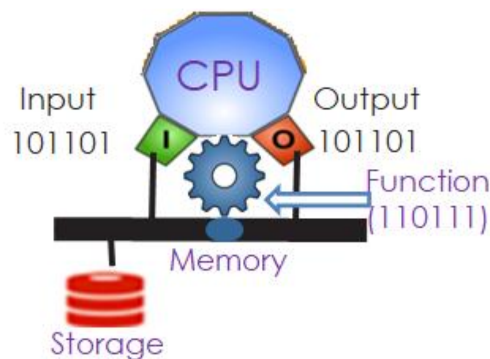


Figure 1: Stored Program Implementation of the Turing Machine model (symbolic computing)

Figure 2 shows the diagram of a computing device based on the Turing Machine schema. While the data are the labels describing the observables defined as sequences of symbols, data structures define the relationships between data items. A state vector defines a named set of data or data structures. Control Processor (a human operator or an "automaton") sets up the hardware (CPU, Memory) and Software (a program that defines the algorithm and data). The program specifies the operations to be performed on the data structures that transform the state vector from the current state to the next. Information processor executes the operations on the state vector in sequence.

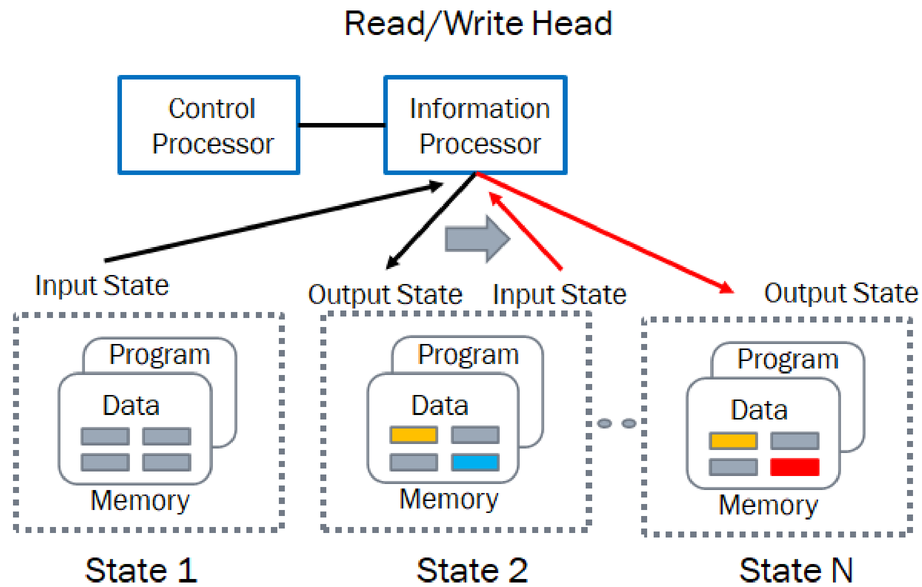


Figure 2: Turing Machine Schema and operations.

State history contains information as the difference between the state vectors. In general, the software can be defined as the symbolic component of a computer system, which determines the functioning of this system.

In general, there are two types of software:

1. *Authentic computer software* is a system of computer programs (functional requirements).
2. *Extended computer software* consists of a system of computer programs and additional symbolic components of the computer system used for determining the functioning of the computer system (non-functional requirements).

There are also other classifications of computer software. The target classification consists of the following categories:

1. *System computer software*, or platform software, consists of programs the goal of which is to control hardware functioning (often represented as IaaS, infrastructure as a service).
2. *Application computer software* consists of programs the objective of which is to provide tools for the user for solving various problems (often represented as PaaS, platform as a service).

In turn, the system computer software consists of the following parts:

1. An operating system
2. Firmware
3. Devices drivers
4. Utilities

Application computer software consists of the following categories:

1. *Acquired software*
2. *User-written software*

The justification classification of computer software consists of the following categories:

1. *Affirmative computer software* consists of programs the goal of which is to control hardware functioning.
2. *Malicious computer software*, or *malware*, is aimed at harming the computer system, disrupting its normal functioning, stealing information, and other mischievous actions.

Software design characteristics include three distinct components:

1. Functional Requirements
2. Non-Functional Requirements
3. Stability, Scalability, Safety, and Security Maintenance Using Best Practices

Software architecture is the structure of a software system which includes various sub-systems, which interact with each other to exchange information. The hardware architecture is the structure of hardware components providing the metabolism for the execution of software components housed in them.

An approach to unveil the operational nature of computer and network software and hardware is made in the operator theory of computation (Burgin and Dodig-Crnkovic, 2020). The operator theory suggests describing computational software, hardware, and the whole computing systems by their functioning, i.e., by their *pure external structures* in the sense of the general theory of structures (Burgin, 2012). In this context, the system of all processes generated by a computing system Q is described as an operator A_Q, while each such process (transformation) is an application of operator A_Q to the input objects. For instance, a Turing machine operator describes the transformation of the input words into the output results by a Turing machine. At the same time, a structural machine operator describes the transformation of the input words into the output results by a structural machine.

Symbolic computing is exploited to execute those tasks that can be easily described by a list of formal, mathematical rules or a sequence of event-driven actions such as modeling, simulation, business workflows, interaction with devices, etc. In addition, symbolic computing is also used to execute an algorithm that models the neural network or connectionist computing (known as sub-symbolic computing) used to analyze vast amounts of data generated in executing these tasks and gaining an understanding of the hidden correlations and insights associated with tasks that are easy to do “intuitively”, that feel automatic, but are hard to describe formally or a sequence of event-driven actions such as recognizing spoken words or faces.

Symbolic computing and sub-symbolic computing structures provide a means to encapsulate operational knowledge of a process that specifies the state of a system (containing various entities, their relationships, and behaviors when events change their state) as sequences of symbols and operations on them. This is very similar to how the body and brain function using the two information processing structures, namely the genes and neurons. Genes specify and execute the metabolic processes that

convert energy and matter (metabolism) to build and operate various information-processing structures in the form of five senses. The neurons provide the means to process information received from the senses to convert it into operational knowledge and execute various operations to achieve various operational processes. Figure 3 shows the similarities.

While they are similar in using symbolic, and super-symbolic information processing structures, biological systems have evolved through natural selection to develop a super-symbolic structure in the form of the neocortex and a sense of "self" and its relationship with its environment which enable them to exhibit autopoietic and cognitive behaviors that 4E cognition alone cannot support. The genome of biological systems, is in essence, software that specifies and executes operational processes that support both cognitive and autopoietic behaviors.

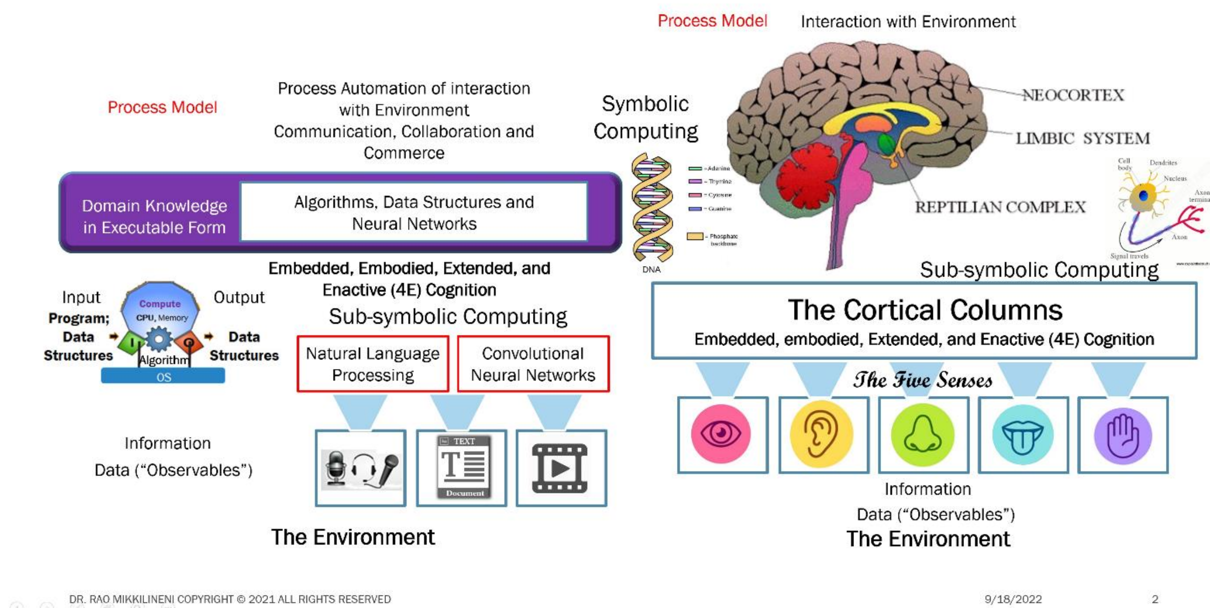


Figure 3: Symbolic and Sub-symbolic computing structures processing information in the physical and digital worlds

There are three foundational shortcomings that prevent Turing Machine-Based computing structures cannot incorporate autopoietic and cognitive behaviors exhibited by biological systems:

1. A digital computing process is executed by several distributed software components using computing resources often owned and managed by different providers and the assurance of end-to-end process sustenance, its stability, safety, security, and compliance with global requirements requires a complex layer of additional processes that increase complexity leading to who manages the manager' conundrum. Any failure in the system requires information access and analysis from multiple sources which results in a reactive approach to fixing the problems.,
2. As Cockshott et al, pointed out in the last paragraph of the last chapter of the book, 'Computation and its Limits', the concept of the universal Turing machine has allowed engineers and mathematicians to create general-purpose computers and "use them to deterministically model any

physical system, of which they are not themselves a part to an arbitrary degree of accuracy. Their logical limits arise when we try to get them to model a part of the world that includes themselves" (Cockshott, et al., 2012) p. 215).

3. Sub-symbolic computing with a neural net computing model provides insights into data, but integrating the new knowledge with other processes is cumbersome if not existent. In this paper, we discuss how super-symbolic computing integrates knowledge from both symbolic and sub-symbolic computing structures. The black box nature of neural networks has to be integrated with other process models that execute various autopoietic and cognitive behaviors that integrate the computer and the computed seamlessly. This requires super-symbolic computing which integrates the symbolic and sub-symbolic processes just as the neocortex does in the human brain.

In the next section, we discuss various theoretical models depicting the software systems and pave the path to infusing autopoietic and cognitive behaviors into current-generation software systems.

Software Systems and Their Theoretical Models

Authentic computer software is the core of a computer system and the principal goal of software development processes. It is possible to separate three stages of software development:

- System design
- Programming and documenting
- Testing, validation, and adaptation

Thus, we come to programs as the central component and predominant objective of software development. As the result, the main efforts of computer scientists have been pointing toward the development of theoretical tools for modeling computer programs as specific algorithms.

In general, the concept of a program is more encompassing than the concept of a computer program. Namely, we have the following definition.

Definition 3.1. A *program* is a compact description of a process or a system of processes.

This characterization allows the specification of the concept of a computer program.

Definition 3.2. A *computer program* is a compact description of a process or a system of processes performed by a computer system under the control of this program.

Consequently, a computer program is a kind of algorithm, or more exactly, of the symbolic representation or description of algorithms (Burgin, 2005). Thus, the mathematical models of algorithms are theoretical models of computer programs.

Various types of such models have been elaborated by computer scientists and mathematicians. The most popular models are:

- Turing machines
- neural networks
- partial recursive functions

- formal grammars
- finite automata

In general, there are three major methods for the development of modeling algorithms and programs:

- *pure generation* means that the new model is created from scratch
- *extension* means that a new model is created by taking an existing model and its extension
- *crossbreeding* means that a new model is created by taking two or more existing models and combining them

The theory of algorithms shows that purely created models are oriented at definite forms of computation and thus, performing well in their problem domain, they have many limitations when it comes to solving problems outside their domain.

As a result, researchers started devising other types of models of algorithms and programs. One of the techniques for doing this was the extension of the existing models. Let us consider examples of such models.

One of the first models of algorithms and programs was recursive functions introduced by Gödel in 1934 (Gödel, 1934). However, it was demonstrated that this model did not represent some important algorithms, and to eliminate this deficiency, Kleene extended recursive functions to partial recursive functions by adding operation minimization (Kleene, 1936).

The most popular model of algorithms and programs was an ordinary Turing machine with one linear tape and one operating head constructed by Turing in 1936 (Turing, 1936). Later researchers extended this model in a different way creating a variety of Turing machines:

- multi-head Turing machines were obtained by adding to the ordinary Turing machine additional heads
- multi-tape Turing machines were obtained by adding to the ordinary Turing machine additional heads and tapes
- Turing machines with *n*-dimensional tapes were obtained by adding to the ordinary Turing machine *n*-dimensional tapes
- nondeterministic Turing machines were obtained by making rules of Turing machines more flexible
- probabilistic Turing machines were obtained by making rules of Turing machines random and assigning probabilities to them (Leeuw, et al, 1956)
- alternating Turing machines are nondeterministic Turing machines with special restrictions (Chandra and Stockmeyer, 1976; Kozen, 1976)
- reflexive Turing machines were obtained by adding rules for changing programs of Turing machines (Burgin, 1992)
- Turing machines with oracles were obtained by adding an additional component called an Oracle to Turing machines and adding rules for interaction with the Oracle (Turing, 1939)
- inductive Turing machines were obtained by adding two additional tapes to the working tape of an ordinary Turing machine and adding rules for interaction with these tapes as well as the new rules of the second order (metarules) for obtaining the result (Burgin, 2010).

Another technique for building new models of algorithms and programs was crossbreeding of the existing models.

An interesting computational crossbreed is a relational machine, which is a Turing machine augmented with a relational memory and operations with relations (Abiteboul and Vianu, 1991). Relational memory can store a set of relations of certain arities[‡]. Some of them are input relations while others are output relations. In addition to the conventional operations of a Turing machine such as changing the internal state, writing on and reading from the tape, and moving the head on the tape, the machine can perform operations with relations. Each such operation is counted as one parallel step of the computation.

Another interesting computational crossbreed is a Neural Turing machine (NTM), which is a recurrent neural network model augmented with a Turing machine (Graves, et al, 2014). These machines amalgamate the fuzzy pattern-matching capabilities of neural networks with the deterministic strength of programmable computers.

The *multiple crossbreeding* of models of algorithms and programs resulted in the creation of the classes of grammars with prohibition (with correction) and classes of selective machines (Burgin, 2005a; 2021; Carlucci, et al, 2009). At first, we consider grammar with prohibition (with correction).

Definition 3.3. A grammar with prohibition (with correction) G is a formal grammar that consists of two parts: positive grammar and negative grammar.

The difference between positive and negative grammar is their purpose of computation. Positive grammars generate or recognize tentative (or possible) elements of the language. However, it is not assumed that all of them are correct, that is, belong to the language under construction. The goal of negative grammar is to recognize those elements that do not belong to the language under construction, that is, are incorrect. This allows building a language by the procedure where at first, tentative (or possible) elements of the language are extracted, and then the incorrect words are eliminated.

Definition 3.4. The *positive language* $L_+(G)$ of grammar with correction G is the language generated/recognized by the positive grammar of G.

Definition 3.5. The *negative language* $L_-(G)$ of grammar with correction G is the language generated/recognized by the negative grammar of G.

Definition 3.6. The language $L(G) = L_+(G) \setminus L_-(G)$ is the *language of the negative*

Depending on the type of positive and negative grammar, we obtain a variety of grammars with corrections.

- (regular, regular)-grammars have both positive and negative grammars are regular.
- In (regular, context-free)-grammar, the positive processor is regular grammar and the negative processor is context-free grammar.
- In (context-free, regular)-grammar, the positive processor is context-free grammar and the negative processor is regular grammar.

[‡] Arity is the number of arguments or operands taken by a function, operation or relation in logic, mathematics, and computer science.

- In (regular, unrestricted)-grammar, the positive processor is a regular grammar and the negative processor is an unrestricted grammar.
- In (context-free, context-free)-grammars have both positive and negative grammars are context-free.
- In (unrestricted, context-free)-grammar, the positive processor is unrestricted grammar and the negative processor is context-free grammar.
- (unrestricted, unrestricted)-grammars have both positive and negative grammars are unrestricted.

In a similar way, selective machines were obtained by computational crossbreeding.

Definition 3.7. A *selective machine M* is a machine (automaton) that has positive and negative processors.

All processors are autonomous.

In what follows, we consider only selective machines with accepting, or what is the same, recognizing processors. Selective machines with other types of processors are studied elsewhere.

Definition 3.8. *Positive processors* are (formal) automata, which accept/recognize words.

Negative processors function in the same way but their results are utilized differently by the selective machine.

Definition 3.9. *Negative processors* are also (formal) automata, which accept/recognize words.

The difference between positive and negative processors is their purpose of computation. Positive processors accept or recognize tentative (or possible) elements of a language. However, it is not assumed that all of them are correct (belong to the language under construction). The goal of negative processors is to recognize those elements that do not belong to the language under construction, that is, are incorrect. This allows building a language by the procedure where at first, tentative (or possible) elements of the language are extracted, and then the incorrect words are eliminated.

We remind that a language L is accepted or recognized by an automaton (machine) M if this automaton accepts all words from L and only these words. It is denoted by L_M or $L(M)$ and is also called the language of the machine M.

Definition 3.10. The *positive language* $L(M_P)$ of M is the language accepted/recognized by positive processors of M.

Definition 3.11. The *negative language* $L(M_N)$ is the language rejected/eliminated/prohibited by negative processors of M.

Definition 3.12. The language $L(M) = L(M_P) \setminus L(M_N)$ is the *language of the selective machine M*.

(m, n)-selective machines have m positive processors and n negative processors. Let us look at some (1, 1)-selective machines, which are simply called selective machines

- FA/FA is the class of all *finite selective machines*, or FF-selective machines, in which both positive and negative processors are finite automata.

- FA/PA is the class of all FP-selective machines, in which the positive processor is a finite automaton and the negative processor is a pushdown automaton.
- FA/TM is the class of all FT-selective machines, in which the positive processor is a finite automaton and the negative processor is a Turing machine.
- FA/SITM is the class of all FSI-selective machines, in which the positive processor is a finite automaton and the negative processor is a simple inductive Turing machine.
- TM/TM is the class of all *Turing selective machines*, or TM-selective machines, in which both positive and negative processors are Turing machines.
- TM/FAis the class of all PF-selective machines, in which a positive processor is a Turing machine and the negative processor is a finite automaton.
- TM/PA is the class of all TP-selective machines, in which a positive processor is a Turing machine and the negative processor is a pushdown automaton.
- TM/ITM is the class of all TI-selective machines, in which a positive processor is a Turing machine and the negative processor is an inductive Turing machine.
- ITM/ITM is the class of all *inductive selective machines*, or SI-selective machines, in which both positive and negative processors are inductive Turing machines.

The most comprehensive model of algorithms and programs is a structural machine (Burgin 2020). First, let us remind the definition of a structural machine.

A structural machine *M* is an abstract automaton, which works with structures of various types. The machine has three major components:

1. The *control device* C_M regulates the state of the machine *M*
2. The (entire) *processor* P_M performs transformation of the processed structures
3. The *functional space* Sp_M contains input, output, and processed structures

The actions (operations) of the entire processor PM depend on the state of machine M and the state of the processed structures. The entire processor consists of one or several unit processors. When a structural machine is considered only as a theoretical model, it is possible that the entire processor contains infinitely many unit processors.

In addition, unit processors can have nested processors that work as subroutines in software systems or natural neurons in the brain. For instance, some researchers suggest that there can be a deep network within a single neuron (cf., for example, (Cepelewicz, 2020)).

Unit processors can move in the processing space performing operations with structures in their neighborhoods according to the rules of their structural machine. Unit processors can function in parallel mode with centralized control. When the structural machine has a distributed control device, which consists of several unit control devices, the unit processors of this machine can function in two modes: clusterized concurrency and total concurrency. Functioning in clusterized concurrency, all unit processors of the structural machine are divided into several groups (clusters) and each group works with its own control device. Functioning in total concurrency, each unit processor has its individual control device, which allows independent operation from other unit processors. As the result, the

architecture of a structural machine provides considerable flexibility, adaptivity, and computational power.

Unit processors of one structural machine can be of different types and categories. For instance, it is possible that one unit processor is a Turing machine, another unit processor is a neural network, while the third one is a cellular automaton and one more unit processor is an inductive Turing machine.

In addition, different unit processors of one structural machine can work in different algorithmic modes. Some of them can perform subrecursive computations, others can compute in the mode of recursive algorithms while the third group can algorithmically function in the super-recursive, e.g., inductive fashion.

An important component of a structural machine - the functional space SpM consists of three components:

1. The input space InM , which contains the input structure(s).
2. The output space OutM , which contains the output structure(s).
3. The processing space PSM , in which the input structure(s) is transformed into the output structure(s), which form the results of computation of a structural machine.

It is often assumed that all structures – the input structures, the output structures and all processed structures – have the same type.

Now we can show that all considered models of abstract automata are special cases of structural machines while descriptive algorithms, such as recursive functions, can be performed by an appropriate structural machine. The same is true for other models of abstract automata, algorithms and programs.

A finite automaton is the simplest case of structural machines that have one processor working with symbols from some alphabet.

A Turing machine with one head and one linear tape is a structural machine that has one processor, which works with linear structures by the rules defined in this Turing machine.

A Turing machine with n heads and one linear tape is a structural machine that has n processors, which work with linear structures by the rules defined in this Turing machine.

A Turing machine with n heads and n linear tapes is a structural machine that has n processors, which work with linear structures by the rules defined in this Turing machine.

A Turing machine with one head and one n-dimensional tape is a structural machine that has one processor, which works with n-dimensional rectangular arrays by the rules defined in this Turing machine.

A Turing machine with n heads and n m-dimensional tapes is a structural machine that has n processors, which work with n-dimensional rectangular arrays by the rules defined in this Turing machine.

A neural network is a structural machine in which processors are artificial neurons, which work with numbers (Burgin, 2005).

It is possible to compute any partial recursive function because such a function can be computed by a Turing machine with one head and one linear tape (Rogers, 1987) while, as it is explained above, it is a particular case of structural machines.

The functioning of a formal grammar can be simulated by a Turing machine with one head and one linear tape (Burgin, 2005) while, as it is explained above, it is a particular case of structural machines.

A nondeterministic Turing machine is a structural machine in which its processor works as a nondeterministic Turing machine.

A probabilistic Turing machine is a structural machine in which its processor works as a probabilistic Turing machine.

An alternating Turing machine is a structural machine in which its processor works as a non-deterministic Turing machine with alternation rules.

A reflexive Turing machine can be simulated by a structural machine that has two processors each of which is working as a Turing machine.

Turing machines with an oracle can be simulated by a structural machine that has two processors one of which is working as a conventional Turing machine while the other one performs the functions of the oracle.

An inductive Turing machine is a structural machine in which its processor works as a Turing machine in the inductive mode (Burgin, 2005).

A relational machine is a structural machine that has two processors one of which is working as a conventional Turing machine while the other one performs operations with set-theoretical relations.

A Neural Turing machine is a structural machine that has two processors one of which is working as a conventional Turing machine while the other one works as a neural network.

A selective machine is a structural machine in which processors have two types – positive and negative.

The functioning of grammar with prohibition (correction) can be simulated by a selective machine, which is, as it is explained above, a structural machine.

Operational Aspects of the General Theory of Information

It is possible to delineate three operational aspects of the General Theory of Information

- GTI as a source of operational information
- Operational countenance of information
- Operational models of information

The GTI explains not only what information is but also how it functions and how to operate with information.

First, the GTI tells (in the Ontological Principle O4) that to operate with information and use it in the material world, information must be represented by physical systems. In addition, information has a

material carrier (in the Ontological Principle O3), which also plays a significant role in the functioning of information in the material world.

Second, the GTI tells (in the Ontological Principle O4) that to operate with information and use it in the mental world, information must be represented by mental systems.

In addition, information has a mental carrier (in the Ontological Principle O3), which also plays a significant role in the functioning of information in the mental world.

Third, the GTI asserts (in the Ontological Principle O5) that interaction is the necessary condition for operation of information and its utilization.

As the result, the GTI presents information in the form, characteristics of which are distinctive for the concept of operator. Let us remind this concept in the most general form developed in (Burgin and Brenner, 2017).

In physics, an operator is a function over a space of initial physical states to the space of final states. In classical mechanics, the movement of a particle (or a system of particles) is completely determined by the Lagrangian or equivalently the Hamiltonian operator of a system.

Operators in classical mechanics are related to symmetries which reflect invariance of motion with respect to a coordinate (Noether's theorem). So, translational, rotational, Galilean transformation, parity and T-symmetry, each is connected with a specific classical mechanic operator. Operators in quantum mechanics are integral part of the formulation of QM. Thus position, momentum, kinetic energy, angular momentum, spin, and Hamiltonian are expressed as operators in QM.

An example from quantum physics is S-matrix (scattering matrix), which denotes an operator that describes the process of transfer of a quantum-mechanical system from the initial state to the final one as a result of a scattering. Taking the set of quantum numbers describing the initial and final states, the scattering amplitudes form a table, which is called the scattering matrix S.

In quantum chemistry according to Levine, an operator is defined as ``a rule that transforms a given function into another function'' (Levine, 1991). The differentiation operator d/dx is an example of operator that transforms a differentiable function $f(x)$ into another function $f'(x)$. Other examples include integration, the square root, and so forth. Numbers can also be considered as operators (they multiply a function).

Operators are widely used in computer programming as well. For example, the Boolean operators, AND, OR, NOT (or AND NOT), and NEAR, with variations such as XOR, are used in logic gates. Furthermore, assignment operators, which assign a specified value to another value and relational operators, which compare two values are widely used in computer programming.

The ontological operator theory of Burgin and Brenner provides the most encompassing definitions of operators and related concepts. Here we use definitions from this theory (Burgin and Brenner, 2017).

Definition O1: An operator is an object (system) that operates, i.e., performs operations on, some objects, systems or processes, which are called operands of this operator.

It is possible to consider three form-oriented types of operators:

- A *symbolic operator* is an operator that has a symbolic form.
- A *material operator* is an operator that has a material form.
- A *mental operator* is an operator that is a part (element) of mentality.

Mathematical operators in functional analysis and mathematical operator theory are symbolic operators.

People and computers are material operators.

Mental operational schemas and algorithms are mental operators (Piaget, 1952; Burgin, 2006; Burgin and Mikkilineni, 2021).

There are also three form-oriented categories of operators:

- A *social operator* is an operator that works (functions) in society.
- A *natural operator* is an operator that works (functions) in nature.
- A *technological operator* is an operator that works (functions) in an artificial world created by people, which includes technology and has been created by technology.

Politicians and businessmen are social operators.

Natural computing systems are natural operators (Dodig-Crnkovic, 2020; Dodig-Crnkovic and Giovagnoli, 2012).

Computers are technological operators.

However, the definition of operators needs some additions. To be complete, Definition O1 demands the following definition.

Definition O2: An operand is an object, system or process operated by an operator.

This definition gives us three target-oriented categories of operators:

- A *socialized operator* is an operator that works with/on social structures as operands.
- A *symbolized operator* is an operator that works with/on symbols (symbolic structures) as operands.
- A *naturalized operator* is an operator that works/on with natural objects (systems) as operands.

Politicians are socialized operators.

Computers are symbolized operators.

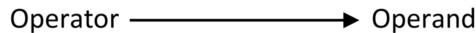
Genes are naturalized operators.

Definitions O1 and O2 show that being an operator or an operand is a role and a characteristic of a system/object. One and the same system/object can be an operator in some situations and an operand

in other situations. In a similar way, a system/object can be an operator with respect to some systems and not an operator with respect to other systems.

Definitions O1 and O2 express the fundamental dyadic relation between operators and their operands, which is actualized in the form of the operator triad:

Operation/function



This diagram presents an operation as a component of an operator triad.

The operator triad is a special case of the basic fundamental triad (Burgin, 2011). In the symbolic representation, it has the form

$$(\mathrm{Op}, \mathit{on}, \mathrm{Od})$$

where Op is an operator, *on* is an operation and Od is an operand.

To construct a general mathematical operator theory in some domain, for example, in the realm of computations, it is necessary to organize the multiplicity of relevant operands in the form of an operating space, i.e., the space that is transformed by an operator.

In this context, the key formal model of an operator Op also has the form of the basic fundamental triad

$$\mathrm{Op} = (D, \mathit{on}, C)$$

where D = D(Op) is the domain of the operator Op, i.e., a space that contains all objects that are operands of this operator

on is the operation that the operator Op performs

C = D(Op) is the codomain of the operator Op, i.e., a space that contains all objects that are results of this operator

Together the domain D and codomain C form the operating space of the operator Op.

An arbitrary operator A is not necessarily defined for all elements from its domain D(A). The subspace (subset) of D(A) where A is defined is called the definability domain and denoted by DD(A). For instance, taking a Turing machine $T_{\varnothing}$ that works with words in the alphabet {0, 1} but never halts independently of its input, we see the domain $D(T_{\varnothing})$ is the set of all words in the alphabet {0, 1} while the definability domain $DD(T_{\varnothing})$ is the empty set $\varnothing$.

In a similar way, the range R(A) of an operator A, i.e., the set of all elements that are values of A, can be only a part of its codomain C(A). For instance, the codomain $C(T\varnothing)$ is the set of all words in the alphabet {0, 1} while the range $R(T\varnothing)$ is the empty set $\varnothing$.

Having the general definition of operator, we can explain why information by its nature is inherently connected to operators. To do this, let us consider the definition of information, which is given in the second ontological principle of the GTI, which has several forms. Here the most general form is given.

Ontological Principle O2 (the *General Transformation Principle*). In a broad sense, *information* for a system *R* is the potentiality/cause of formations and transformations (changes) in the system *R* or of their prevention in the system *R*.

Thus, we may understand information in a broad sense as a capacity (ability or potency) of things, both material and abstract, to change other things. Information exists in the form of *portions of information*. Informally, a portion of information is such information that can be separated from other information. Information is, as a rule, about something. What information is about is called an *object* of this information.

The Ontological Principle O2 has several consequences. First, it demonstrates that information is closely connected to transformation. Namely, it means that information and transformation are functionally similar because they both point to changes in a system (Burgin, 1998/1999). At the same time, they are different because information is potency for (or in some sense, cause of) change, while transformation is the change itself, or in other words, transformation is an operation, while information is what induces this operation.

Second, the Ontological Principle O2 explains *why* information influences society and individuals all the time, as well as why this influence grows with the development of society. Namely, reception of information by individuals and social groups induces transformation. In this sense, information is similar to energy. Moreover, according to the Ontological Principle O2, energy is a kind of information in a broad sense. This well correlates with the Carl Friedrich von Weizsäcker's idea (Weizsäcker, 1974; 1985) that *energy might in the end turn out to be information*.

Third, the Ontological Principle O2 makes it possible to separate different kinds of information. For instance, people, as well as any computer, have many kinds of memory. It is even supposed that each part of the brain has several types of memory agencies that work in somewhat different ways, to suit particular purposes (Minsky, 1986). It is possible to consider each of these memory agencies as a separate system and to study differences between information that changes each type of memory. This might help to understand the interplay between stability and flexibility of mind, in general, and memory, in particular.

In essence, we can see that all kinds and types of information are encompassed by the Ontological Principle O2.

However, the common usage of the word *information* does not imply such wide generalizations as the Ontological Principle O2 expresses. Thus, we need a more specific and consequently, more restricted theoretical meaning because an adequate theory, whether of the information or of anything else, must be in a significant accord with our common ways of thinking and talking about what the theory is about, else there is the danger that theory is not about what it purports to be about. To achieve this goal, we use the concept of an *infological system* IF(*R*) of the system *R* for the information definition. It is done in two steps. At first, we make the concept of information relative and then we choose a specific class of infological systems to specify information in the strict sense. That is why it is impossible and, as well as, counterproductive to give an exact and thus, too rigid and restricted definition of an infological system. Indeed, information is a very rich and widespread phenomenon to be reflected by a restricted rigid definition (cf., for example, (Capurro, *et al*, 1999; Melik-Gaikazyan, 1997)).

The concept of infological system plays the role of a free parameter in the general theory of information, providing for representation of different kinds and types of information in this theory. That is why the concept of *infological system*, in general, should not be limited by boundaries of exact definitions. A free parameter must really be free. Identifying an infological system IF(*R*) of a system *R*, we can define information relative to this system. This idea is expressed in the following principle.

Ontological Principle O2g (**the *Relativized Transformation Principle***). *Information* for a system *R relative to the infological system* IF(*R*) is a capacity to cause changes in the system IF(*R*) or of their prevention in the system IF(*R*).

As a model example of an infological system IF(*R*) of an intelligent system *R*, we take the system of knowledge of *R*. In cybernetics, it is called the *thesaurus* Th(*R*) of the system *R*. Another example of an infological system is the memory of a computer. Such a memory is a place in which data and programs are stored and is a complex system of diverse components and processes.

Elements from IF(*R*) are called *infological elements*.

There is no exact definition of infological elements although there are various entities that are naturally considered as infological elements as they allow one to build theories of information that inherit conventional meanings of the word *information*. For instance, knowledge, data, images, algorithms, procedures, scenarios, ideas, values, goals, ideals, fantasies, abstractions, beliefs, and similar objects are standard examples of infological elements.

When we take a physical system *D* as the infological system and allow only for physical changes, we see that information with respect to *D* coincides with energy.

Taking a mental system *B* as the infological system and considering only mental changes, information with respect to *B* coincides with mental energy.

These ideas are crystallized in the following principle.

Ontological Principle O2a (**the *Special Transformation Principle***). *Information in the strict sense* or *proper information* or, simply, *information* for a system *R*, is a capacity to change or to prevent changes of the structural infological elements from an infological system IF(*R*) of the system *R*.

As changes are performed by operators, this reflects the operational nature of information. So, it was organic that the mathematical models of information have been developed in the form of functors in the categorical setting forming information algebras, calculi and topological spaces as well as operators in functional spaces. These models create the base of the operational mathematical theory of the GTI.

The first type of models is built in abstract categories, which allow the development of flexible tools for the studies of information and its flow, as well as of computers, networks and computation. There are two types of information dynamics representations in categories: the categorical representation and functorial representation (Burgin, 2010a; 2011a). Categorical representations of information dynamics preserve internal structures of information spaces associated with infological systems as their state or phase spaces. Functorial representations of information dynamics preserve external structures of information spaces associated with infological systems as their state or phase spaces. This provides a base for analyzing physical and information systems and processes by means of categorical structures and methods.

The categorical representation of information dynamics preserves internal structures of information spaces associated with infological systems as their state or phase spaces. In it, portions of information are modeled by categorical information operators (Burgin, 2010a). The functorial representation of information dynamics preserves external structures of information spaces associated with infological systems as their state or phase spaces. In it, portions of information are modeled by functorial information operators (Burgin, 2011a).

The second type of models represent information that acts on knowledge, bringing new and updating existing knowledge, is of primary importance to people. It is called *epistemic information*, which is studied based on the general theory of information and further developing its mathematical stratum. Portions of epistemic information are modeled/represented by epistemic information operators acting in spaces of knowledge, which are represented by a formal construction called a Mizzaro space (Burgin, 2011; 2014). These spaces consist of knowledge items often unified by structural relations.

Conversion of Information into Operational Knowledge

Although the information is operational by its nature, its realization demands the representation of information in such a way that it can be converted into operational knowledge. There are different types and forms of operational knowledge: operators, algorithms, programs, plans, procedures, methods, operational schemas, instructions, etc. The general definition of an operator allows the representation of all types and forms of operational knowledge by operators. Naturally, there are operators in mentality as well as physically presented operators. For instance, a software system is a physically presented operator. As we are mostly interested in software development, we analyze conversion information into physically presented operators.

The process of construction of a physically presented operator consists of several stages.

At first, based on different information, a decision is made on what the constructed operator, e.g., the planned software system, must do or in other words, what is the goal of the tentative operator functioning. This decision is based on different information. Those who make this decision already have some of the necessary information while other information must be obtained.

The next stage is the evaluation if it is possible and practical to construct an operator that solves the necessary problem (achieves the necessary goal).

One more stage includes the evaluation of the necessary resources such as the initial data, knowledge, and information, the sufficient means (tools) for construction, and the necessary time to do this. Note that time is a very important resource. Time restriction can make a potentially solvable problem practically unsolvable.

It is possible that after finishing the third stage, it becomes necessary to go once more to the second stage or even to the first stage changing the problem and or goals. This can create a cycle, after finishing which the process goes to the fourth stage, where the description of the forthcoming operator, e.g., the specification of the software system, is elaborated.

At the fifth stage, the plan of the operator construction is elaborated. Often this plan is further developed to get the procedure of the operator construction. It is possible to call such plans and

procedures by the name *metaoperator*. We remind that a metaoperator is an operator that acts on other operators, that is, an operator, operands of which are also operators.

At the sixth stage, the necessary physically represented operator, for example, in the form of a software system, is constructed.

However, this is not the end because it is necessary to show that the constructed operator has all the necessary properties, that is, it is correct. To demonstrate this, such methods as testing, logical verification, and inspection or auditing are utilized (Burgin and Debnath, 2012).

It is necessary to understand that each stage, at first, is performed in mentality resulting in mental operational knowledge, and then using information from this mental knowledge, physical operational knowledge in the form of algorithms, programs, and software and hardware systems is elaborated.

The GTI explains that to perform all operations at each of these stages, it is necessary to choose or to create efficient mental and physical representations of the used and produced information. For instance, programming is the conversion of operational information in mentality into a physical representation of information as symbolic operators in the physical form of programs. Computer programs are particular cases of operational knowledge in the physical form.

The GTI makes clear that the efficiency of the physical representation of operational knowledge and information in general, e.g., symbolic operators, and of software, in particular, is a vector characteristic, which consists of three components:

- *Construction efficiency* is measured by complexity and other characteristics of the construction of the physical representation
- *Utilization efficiency* is measured by complexity and other characteristics of the utilization of the physical representation
- *Management efficiency* is measured by complexity and other characteristics of the management of the physical representation

To work with computers, people constructed programming languages, which allow transmitting operational information to computers. Software of computers, networks, and cell phones consists of symbolic operators in the physical form of programs, which, as it was explained before, is operational knowledge.

With respect to software development, construction efficiency means easiness, completeness and resourcefulness of the process of software development. In other words, construction efficiency is also a vector of three components.

Utilization efficiency with respect to software development is also a vector with such components as functionality, reliability, usability, effectiveness, flexibility, portability, integrity, and usefulness of the constructed software system.

Management efficiency with respect to software development is also a vector with such components as maintainability, security, and self-restoration.

Note that it is necessary not only to identify characteristics of the efficiency of the physical representation of operational knowledge and information but also to develop measures for the evaluation of efficiency and its components. General approaches to building such measures are described in the axiological principles of the GTI.

The construction of the physical representation of operational knowledge and information in general, e.g., symbolic operators, and software systems in particular usually follows one of the following approaches:

- In the *model-based approach*, a physical representation is constructed mirroring some operational model, requiring correct specification of operators and being open to potential modeling biases.
- In the *design-based approach*, a physical representation is constructed imitating some model design process, requiring efficient operator design methods, and is open to potential design biases.
- In the *problem-based approach*, a physical representation is constructed with orientation on the problem that is being solved, requiring an efficient description of input and clear formulation of goals and being open to potential problem solution biases.

To apply these approaches to software development, it is necessary to specify them taking into account specialized characteristics of software systems. For instance, when the model-based approach is used for the development of software systems, we need to utilize models of software systems some of which are described in Section 3.

New Approach to Building Self-Managing Software Systems

All living organisms exhibit teleonomy, which is the quality of apparent purposefulness and of goal-directedness of structures and functions brought about by natural processes like natural selection. The genome provides the specification of a model for software that encodes the life processes and executes them by building and using the information processing structures that utilize the transformation processes of matter and energy.

The genes that make up the human genome are best viewed as a society (Yanai, I. Martin, L. 2016). The human genome contains about 20,000 genes, each of which specializes in one or more tasks. Genes need to cooperate to build and run a body capable of replicating them. These feats require an intricate organization and fine-tuned division of labor. As Mitchell Waldrop (Waldrop, 1992) pointed out, “DNA residing in a cell's nucleus was not just a blueprint for the cell a catalog of how to make this protein or that protein. DNA was actually the foreman in charge of construction. In effect, DNA was a kind of molecular-scale computer that directed how the cell was to build itself and repair itself and interact with the outside world. Life processes including autopoietic and cognitive behaviors are specified in the genome and are executed using the DNA. Autopoietic behaviors implement the metabolism using the transformation of energy and matter. Replication of cells and their specialization allows the functioning of autonomic components to build and operate a cognitive network that manages both the “self” and its interactions with the environment exchanging information. Life processes including learning from the information received from the five senses and its processing using the cognitive apparatuses in the form of genes and neurons are all encoded in the genome. Replication and metabolism are used to execute

the network of networks where each node executes a process and communicates with other nodes exchanging information. Shared knowledge between the communicating nodes enables the behavioral changes that evolve the system changes. It is not an exaggeration to say that without a genome specification of life processes, there is no natural intelligence.

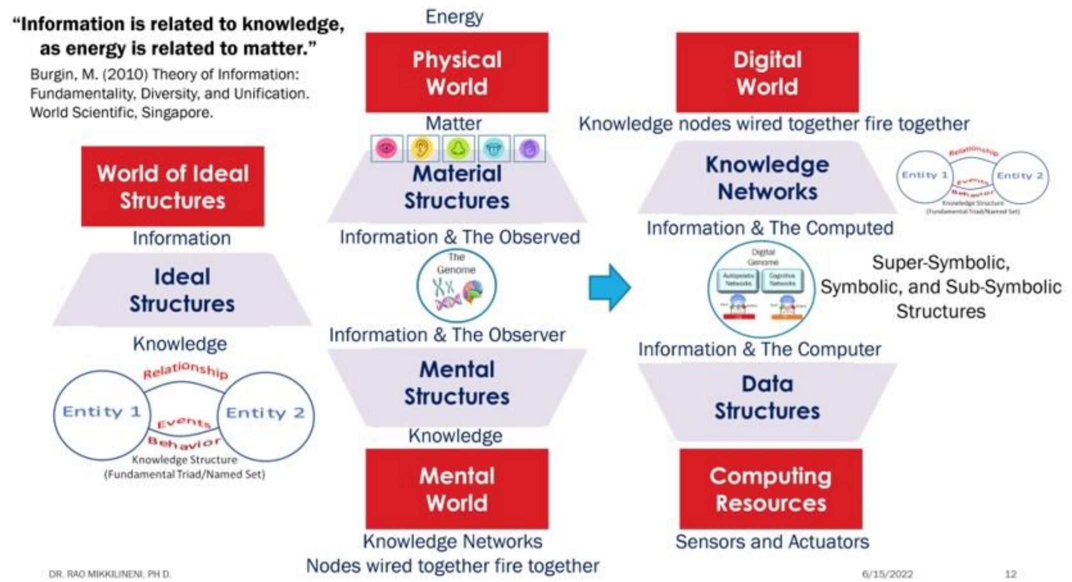


Figure 4: General Theory of Information and the models of autopoietic and cognitive structures in mental and digital worlds

The general theory of Information provides a framework with structural machines, cognizing oracles, and knowledge structures to model autopoietic and cognitive behaviors. The schema and the operations on the schema provide a means to represent process knowledge and execute the processes in a multi-layered network using the fundamental triads or named sets in the form of entities, relationships, and event-induced behaviors of the system. Figure 4 shows the relationship between ideal structures material structures in the physical universe and the mental structures in the mental world are enabled by cognitive and autopoietic behaviors. The picture also shows how to infuse autopoietic and cognitive process execution in the digital world using both symbolic, and sub-symbolic computing structures in a super-symbolic computing structure. Figure 5 shows the schema and operations representing a knowledge structure. The difference between Figures 2 and 4 is that the functional processor in figure 4 operates on knowledge structures specifying a process in the form of various entities, relationships, and their state evolution behaviors in time depending on events that cause changes in any of the states of the entities or their relationships, while the functional processor in Figure 2 operates on the data structures. In addition, the structural machine defined in figure 4 contains three levels of processing structures as discussed in section 3. At the top level, the structural machine control processor manages various autopoietic and cognitive processes implemented as grid automata.

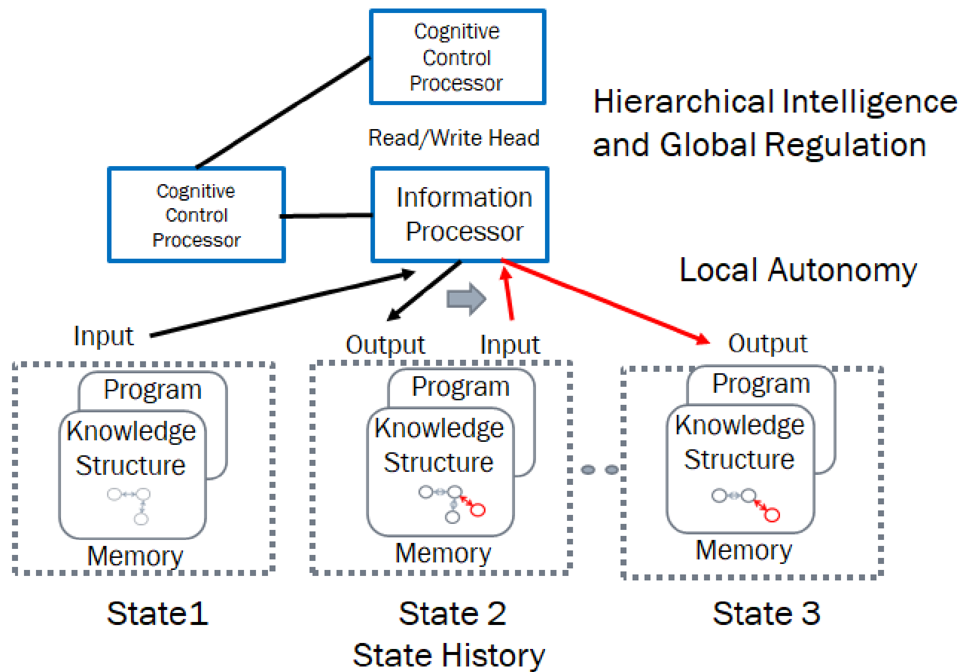


Figure 5: Knowledge Structure Schema and Operations

Figure 6 shows the structural form of the physical Implementation of the operational knowledge network and the knowledge structure. Local, cluster-wide, and global autopoietic and cognitive behaviors are implemented in the three layers using a meta-Knowledge Structure. We define the meta-Knowledge Structure as a computing structure that implements the functional requirements, and non-functional requirements and has the knowledge to use policies and best practices to maintain stability, security, safety, and survival with optimal resource utilization.

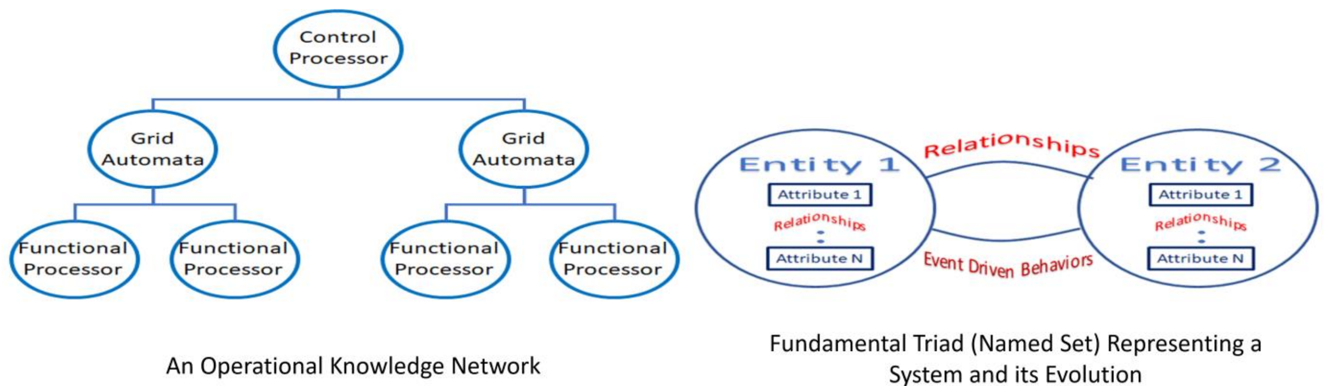


Figure 6: The structural form of the physical Implementation of the operational knowledge network and the knowledge structure

Figure 7 shows the implementation of a meta-Knowledge Structure using conventional infrastructure (e.g., IaaS), middleware (e.g., PaaS), and application software components executing various functions required to execute the process defined as a knowledge structure.

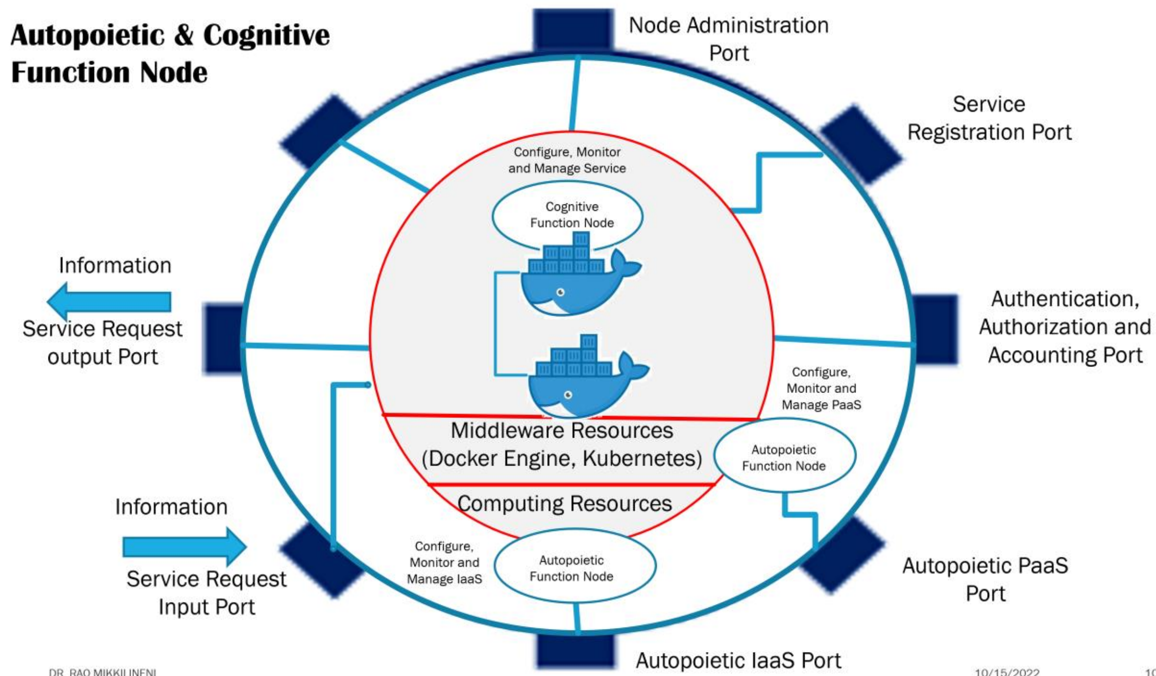


Figure 7: Meta-Knowledge Structure implementation using conventional IaaS, PaaS, and containers.

There are many ways to implement the meta-Knowledge structure using existing symbolic and sub-symbolic computing structures and a variety of communication mechanisms to share information among interacting nodes to create an intelligent knowledge network with autopoietic and cognitive behaviors. Autopoiesis in digital automata involves the knowledge of both how software is used to execute a process involving, various entities, relationships, and their evolution based on event-driven interactions (behaviors) and how to provide the metabolism required to execute myriad components in the system that interact with each other and their environment. Cognition in the digital automata involves the process knowledge to execute the goals of the system and the best practices that assure the successful execution of the goals in the face of fluctuations caused by internal or external events.

Figure 8 shows an implementation based on the meta-knowledge structures presented in the 12th International Congress on Advanced Applied Informatics (Burgin, Mikkilineni 2022). The paper and presentation demonstrate the application of the concepts presented here to infuse autopoietic and cognitive behaviors into an open-source application to provide self-regulation without service disruption in the face of fluctuating demand for or the availability of computing resources.

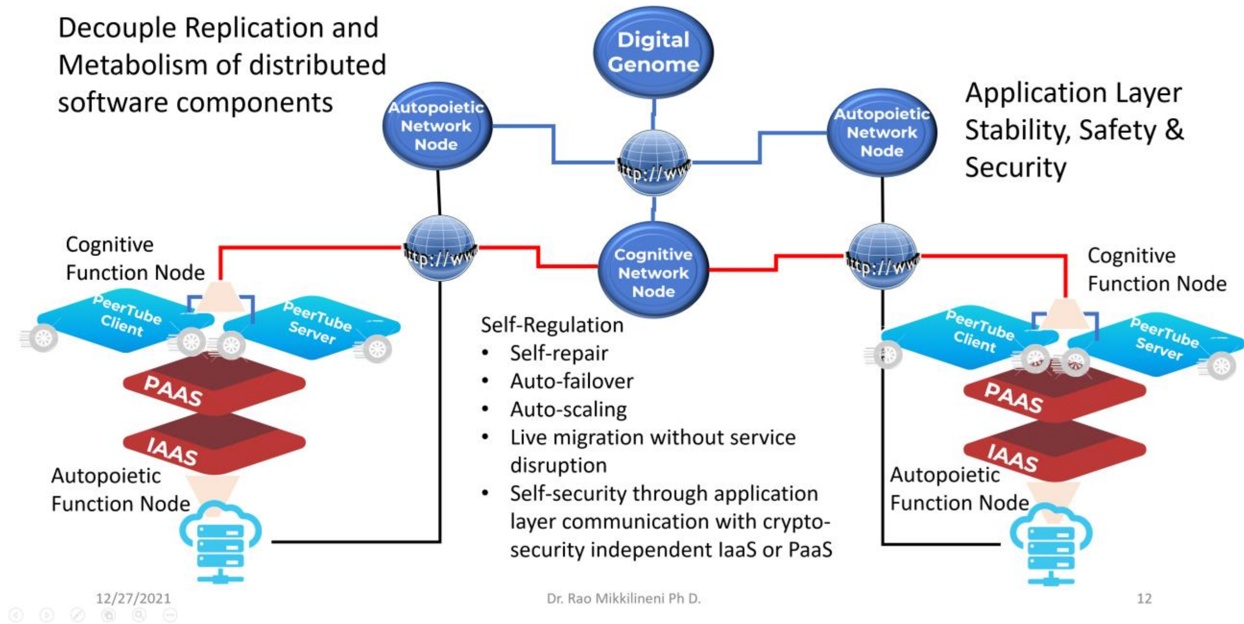


Figure 8: Implementation of an open-source application PeerTube using meta-knowledge structures with infused cognitive and autopoietic behaviors.

The paper and presentation demonstrate the application of the concepts presented here to infuse autopoietic and cognitive behaviors into an open-source application to provide self-regulation without service disruption in the face of fluctuating demand for or the availability of computing resources.

Another implementation (Lee, and Venters, 2022) attempts to use meta-knowledge structures to address the healthcare disease diagnostic process where various entities such as patients, symptoms, diseases, primary care physicians, and specialists interact. When a patient experiences various symptoms, they often, point to multiple diseases as the cause for these symptoms. Medical knowledge already exists about these symptoms and their relationships to various diseases. Primary care physicians (PCP) and various specialists acquire this knowledge through their training in the study and practice of medicine. When patients experience these symptoms with various levels of severity, they go to a PCP who treats them using his knowledge, and when the patients need specialized care from a specialist, they are referred to and treated by the specialists.

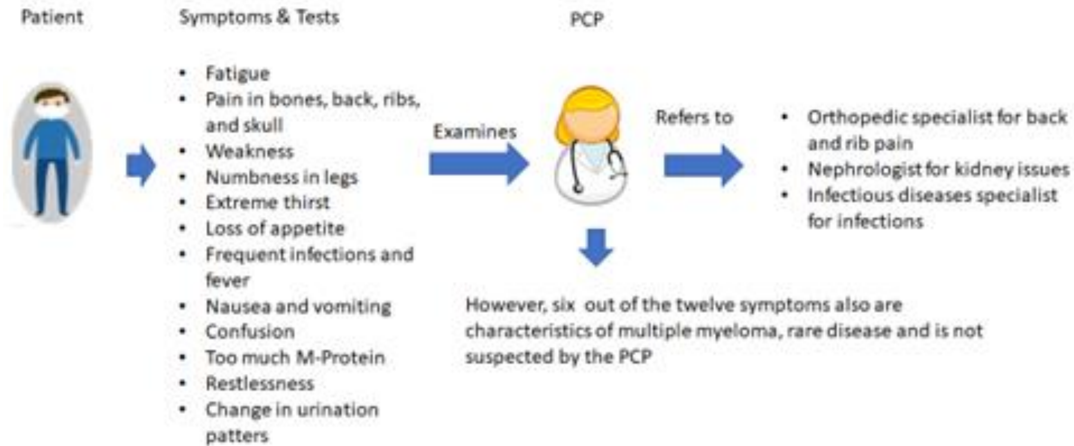


Figure 9: Diagnostic Process

This process has several shortcomings:

1. As the Figure 9 shows, the PCP often misses certain rare diseases and often, it delays the discovery of the rare disease.

2. Often AI is used to detect possible diseases from the symptoms based on big data analytics. However, this approach suffers from the 'Black Box' issue (unknown mechanisms churn out a result from a selected set of inputs). The center for disease control (CDC) suggests that For AI to gain wider interest from clinicians, the way the algorithms arrive at their conclusions needs to be understandable.

3. Most Americans will experience a diagnostic error at least once in their lifetime. Patient deaths due to these errors are estimated at 40,000 to 80,000 per year. Diagnostic errors and other inefficiencies cost the U.S. economy $750 billion each year.

4. Average Time Spent with Patient: 15 minutes. As a result, misdiagnosis and delayed diagnosis are common. Communication issues based on incorrect or missing information cause burnout in healthcare professionals

The solution is a digital genome specification of the diagnostic process based on the meta-knowledge structures capturing the relationships and event-triggered behaviors of the patients, symptoms, diseases, PCPs, and specialists. Figure10 shows the structural schema of an individual health care process. The knowledge about diseases and symptoms and the knowledge about diseases and specialists are obtained from medical ontologies. The meta-knowledge structure captures the behaviors of various patients and the history of events and triggered behaviors. It also provides explainable relationships between symptoms a patient experiences and possible diseases and the specialists that could assist in further diagnosis. In essence, a model-based AI augments current AI results by incorporating big data analytic results along with model-based meta-knowledge structures.

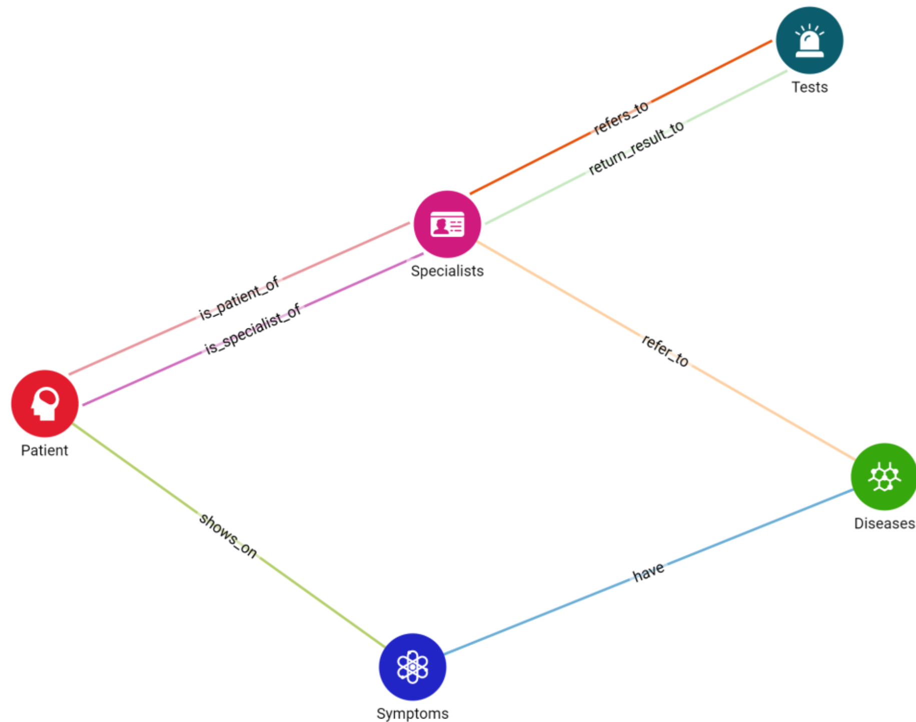


Figure 10: Schema depicting the entities, relationships and events triggering various behaviors

Conclusion

In this paper, we applied the General Theory of Information and various tools derived from it to lay the foundation for software system design, development, execution, and its lifecycle management while integrating current-generation software technologies and processes. As mentioned in this paper, this is very similar to how biological systems evolved their information processing structures to improve system resiliency, and efficiency at scale using a local, cluster, and global intelligence through autopoietic, and cognitive behaviors. In essence, the genome is the quintessential software system that uses replication, and metabolism to build, monitor, and manage 30+trillion cells. Biological intelligence stems from the cells executing various "life" processes specified in the genome using information processing structures (genes, and neurons) that use "replication" and "metabolism." These processes include the knowledge to execute various tasks that use the laws of transformation of energy and matter to create a society of cells that exhibit autopoietic and cognitive behaviors. This knowledge includes how to manage cell behaviors to not only build, and maintain the system with an identity ("self") but also how to interact with the environment to receive information and convert it into executable knowledge that allows the system to model the external world and interact with it with specific goals that are inherited genetically or acquired through the learning process. Replication makes autopoietic behaviors possible. Metabolism allows the transformation of energy and matter to fuel the "life" processes executed by the cells. The transformation of energy and matter allows the formation of cells that, act as a society.

The general theory of information explains importance of having efficient tools for embedding operational information. An important class of such tools is formed by programming languages. As we know, now different stages of software development use languages with different structures

and organization. At the same time, unified tools for software development are provided by the block-schema programming metalanguage (BS-language) (Burgin, 1976; 1978; Burgin, Eggert, 2004).

The general theory of information provides a model for genome specification, and its execution (Burgin 2010; Mikkilineni, 2022). It provides a schema and operations to execute meta-knowledge structures that are analogous to cells using replication, and metabolism to execute autopoietic and cognitive behaviors. While the role of replication and metabolism are discussed in the literature (Dyson 1977), This paper is the first attempt to clarify the role of a schema and operations (Burgin, Mikkilineni 2021) for the meta-knowledge structure that models autopoietic and cognitive behaviors using replication and metabolism. The meta-knowledge structure is like a cell with the knowledge to find and use computing resources to execute the defined processes. The digital genome specifies and executes the society of meta-knowledge structures in the form of a multi-layered knowledge network, where nodes wired together fire together to execute local, clustered, and global autopoietic and cognitive behaviors.

The resulting super-symbolic structure integrates the current generation's symbolic and sub-symbolic structures (Burgin, Mikkilineni 2022a) to enhance information processing capabilities. Hopefully, this will pave the path to enhance our information processing systems with self-regulating properties and build computing structures that are more intelligent than the current state-of-the-art.

References

Abiteboul, S., and Vianu, V. (1991) Generic computation and its complexity, in ``Proc. ACM SIGACT Symp. on the Theory of Computing,'' pp. 209-219.

Abiteboul, S., Papadimitriou, C.H. and Vianu, V. (1997) Fixpoint Logics, Relational Machines and Computational Complexity, Journal of the ACM (JACM), v. 44, No. 1, pp. 30-56

A.J. Albrecht, "Measuring Application-Development Productivity," Programmer Productivity Issues for the Eighties, 2nd ed., C. Jones, ed., IEEE CS, 1981, pp. 3443.

Burgin, M. The Recursion Operator and Representability of Functions in the Block-Scheme Language, *Programming*, 1976, No. 4, pp. 13-23 (Programming and Computer Software, 1976, v. 2, No.4)

Burgin, M.Variables in the Block-Scheme Language, *Programming*, 1978, v. 4, No. 2, pp. 3-11 (Programming and Computer Software, 1978, v. 4, No. 2, pp. 79-85)

Burgin, M.S. (1992) Reflexive Calculi and Logic of Expert Systems, in *Creative processes modeling by means of knowledge bases*, Sofia, pp. 139-160

Burgin, M. Information and Transformation, *Transformacje*, 1998/1999, No.1, pp. 48-53 (in Polish)

Burgin, M. *Super-recursive Algorithms*, Springer, New York/Heidelberg/Berlin, 2005

Burgin, M. Grammars with Prohibition and Human-Computer Interaction, in "Proceedings of the Business and Industry Simulation Symposium," Society for Modeling and Simulation International, San Diego, California, 2005a, pp. 143-147

Burgin, M. *Mathematical Schema Theory for Modeling in Business and Industry*, Proceedings of the 2006 Spring Simulation MultiConference (SpringSim '06), April 2-6, 2006, Huntsville, Alabama, pp. 229-234

Burgin. M. (2010) *Theory of Information*, World Scientific Publishing, New York

Burgin, M. Information Operators in Categorical Information Spaces, *Information,*2010a, *1*, 119 - 152

Burgin, M. Epistemic Information in Stratified M-Spaces, *Information*, 2011, *2*, 697 - 726

Burgin, M. Information Dynamics in a Categorical Setting, in *Information and Computation*, World Scientific, New York/London/Singapore, 2011a, pp. 35 - 78

Burgin, M. *Structural Reality*, Nova Science Publishers, New York, 2012

Burgin, M. Weighted E-Spaces and Epistemic Information Operators, *Information*, 2014, *5*, 357 - 388

Burgin, M. *Theory of Knowledge: Structures and Processes*; World Scientific: New York, NY, USA; London, UK; Singapore, 2016.

Burgin, M. 2017. *The General Theory of Information as a Unifying Factor for Information Studies*: *The noble eight-fold path*, Proceedings, 2017, v. 1, No. 3, 164, 6 p.; doi:10.3390/IS4SI-2017-04044

Burgin, M. (2020) Information Processing by Structural Machines, in *Theoretical Information Studies: Information in the World*, World Scientific, New York/London/Singapore, pp. 323–371

Burgin, M. Modelling distributive computation by selective machines, *International Journal of Parallel, Emergent and Distributed Systems*, v. 36, No.5, 2021, pp. 395 - 411 DOI: 10.1080/17445760.2021.1934837

Burgin, M. and Adamatzky, A. Structural machines and slime mold computation, *International Journal of General Systems*, v. 45, No. 3, pp. 201-224 (2017)

Burgin, M. and Brenner, J.E. *Operators in Nature, Science and Society*, Preprint in General Science and Philosophy 1204.0045, 2012, 45 p. (electronic edition: http://viXra.org/postprints/)

Burgin, M. and Debnath, N.C. *Interplay of Logical Verification and Performance Testing in Software Assurance*, in Proceedings of the 21st International Conference on Software Engineering and Data Engineering (SEDE-2012), ISCA, Los Angeles, California, June 27–29, 2012, pp. 155-160

Burgin, M. and Dodig-Crnkovic, G. *Prolegomena to Operator Theory of Computation*, Information, v. 11, No. 7, 2020, 349; doi:10.3390/info11070349

Burgin, M. and Eggert, P. Types of Software Systems and Structural Features of Programming and Simulation Languages, in "*Proceedings of the Business and Industry Simulation Symposium*," Society for Modeling and Simulation International, Arlington, Virginia, 2004, pp. 177-181

Burgin, M. and Mikkilineni, R. From Data Processing to Knowledge Processing: Working with Operational Schemas by Autopoietic Machines, Big Data Cogn. Comput. 2021, v. 5, 13.

M. Burgin and R. Mikkilineni, (2022) "General Theory of Information Paves the Way to a Secure, Service-Oriented Internet Connecting People, Things, and Businesses," 2022 12th International Congress on Advanced Applied Informatics (IIAI-AAI), pp. 144-149, doi: 10.1109/IIAIAAI55812.2022.00037.

https://youtu.be/PEtrMhzI6yY Video Presented at the conference (Accessed on October 15, 2022)

Burgin, Mark & Mikkililineni, Rao (2022a). Seven Layers of Computation: Methodological Analysis and Mathematical Modeling. Filozofia i Nauka 10:11-32.

Capurro, R., Fleissner, P., and Hofkirchner, W. (1999) Is a Unified Theory of Information Feasible? In *The Quest for a unified theory of information*, Proceedings of the 2nd International Conference on the Foundations of Information Science, pp. 9-30

Carlucci, L., Case, J. and Jain, S. Learning correction grammars, *J. Symb. Logic*, v. 74, pp. 489-516 (2009)

Cepelewicz, J. Hidden Computational Power Found in the Arms of Neurons, *Quanta Magazine*, 2020

Chandra, A.K.., and Stockmeyer, L.J. (1976) Alternation, in *Proceedings of the 23rd Symposium on Foundations of Computer Science*, pp. 98-108

Cockshott, L. M. MacKenzie and G. Michaelson, "Computation and Its Limits," Oxford University Press, Oxford, 2012

Collier, M. and Beel, J. (2018) Implementing Neural Turing Machines, *Artificial Neural Networks and Machine Learning – ICANN 2018*, Springer International Publishing, pp. 94–104

Gödel, K (1934) On Undecidable propositions of Formal mathematical Systems, Lectures given at the Institute for Advanced Studies, Princeton, in *The Undecidable* (Ed. Davis, M.) Raven Press, 1965, pp.39-71

Dodig-Crnkovic, G. Morphological, Natural, Analog and Other Unconventional Forms of Computing for Cognition and Intelligence, Proceedings, 2020, 4

Dodig-Crnkovic, G. and Giovagnoli, R. (Eds.) Natural Computing/Unconventional Computing and its Philosophical Significance, The Society for the Study of Artificial Intelligence and Simulation of Behaviour, 2012

Dyson, G. The Darwin Among the Machines: The evolution of Global Intelligence, Basic Books, New York, 1997.

Graves, A., Wayne, G. and Danihelka, I. (2014) *Neural Turing Machines*, arXiv:1410.5401

Kleene, S. (1936) General Recursive Functions of Natural Numbers, *Mathematische Annalen*, v. 112, No. 5, pp. 727-729

Kozen, D. (1976) On Parallelism of Turing Machines, in *Proceedings of the 23rd Symposium on Foundations of Computer Science*, pp. 89-97

Lee, J., and Venters, (2022) C. Private Communication

de Leeuw, K., Moore, E.F., Shannon, C.E. and Shapiro, N. (1956) Computability by Probabilistic Machines, *Automata Studies*, Princeton University Press, Princeton, N.J. pp. 183-212

Levine, I. N. Quantum Chemistry. Prentice Hall, Englewood Cliffs, NJ, 4th edition, 1991

Melik-Gaikazyan, I. V. (1997) *Information processes and reality*, Nauka, Moscow (in Russian, English summary)

Mikkilineni R. A New Class of Autopoietic and Cognitive Machines. Information. 2022; 13(1):24. https://doi.org/10.3390/info13010024

Minsky, M. (1986) *The Society of Mind*, Simon and Schuster, New York

Piaget, J. (1952) *The Origin of Intelligence in Children*. Trans. M. Cook. New York: International Universities Press

M. Razian, M. Fathian, R. Bahsoon, A. N. Toosi and R. Buyya, (2022). "Service composition in dynamic environments: A systematic review and future directions", Journal of Systems and Software.

Rogers, H. (1987) Theory of Recursive Functions and Effective Computability, MIT Press, Cambridge Massachusetts

Tisdale, S. M. (2015). Cybersecurity: Challenges from a Systems, Complexity, Knowledge Management and Business Intelligence Perspective. *Issues in Information Systems*, *16*(3).

Turing, A. (1936) On Computable Numbers with an Application to the Entscheidungs-problem, *Proc. Lond. Math. Soc.,* Ser.2, v. 42, pp. 230-265

Turing, A. (1939) Systems of Logic Based on Ordinals, *Proc. Lond. Math. Soc.,* Ser.2, v. 45, pp. 161-228

von Weizsäcker, C.F. (1974) *Die Einheit der Natur*, Deutscher Taschenbuch Verlag, Munich, Germany

von Weizsäcker, C.F. (1985) *Aufbau der Physik*, Hanser, Munich, Germany (Eglish translation: *The Structure of Physics*, Springer, Berlin/Heidelberg/New York, 2006)

Yanai, I.; Martin, L. (2016) The Society of Genes; Harvard University Press: Boston, MA, USA, 2016.