

# Algorithmic Control in Concurrent Computations

Mark Burgin

Department of Computer Science  
University of California, Los Angeles  
Los Angeles, CA 90095

**Abstract:** *In this paper, functioning and interaction of distributed devices and concurrent algorithms are analyzed in the context of the theory of algorithms. Although different systems in a network can function independently, their interaction is usually subjected to definite rules. These rules often form algorithms of interaction for distributed systems and devices. The main goal is to understand how control algorithms in distributed systems organize computational processes. Characteristics of distributed computational processes are considered from the algorithmic perspective. Conditions are found when such a network controlled by algorithm functions as a single algorithm and types of such unifying algorithms are determined. All mathematical models of distributed computation, including such popular structures as neural networks, Petri nets, systolic arrays, iterative arrays, and cellular automata, are synthesized in the concept of grid automaton. These automata are used as a basic tool for studying distributed devices and concurrent algorithms.*

**Keywords:** distributed computation, concurrent process, grid array, grid automaton, algorithmic operation

## 1 Introduction

Developing mathematical frameworks that accurately and tractably account for concurrent and distributed processes has been a major goal of researchers in concurrency. As a result, a lot of efforts have been devoted to the construction and study of suitable models. This diversity of models is usually classified with respect to three dimensions [8, 13]: *substantiality*, *structural dimension*, and *time*. In the substantiality dimension, Cleaveland, et al [8] distinguish intensional and extensional models, Sassone, et al [13] separate system and behavior models, while Gupta [10] discerns action based and event based models. In the structural dimension, interleaving and noninterleaving classes of models are treated. In the temporal dimension, models with linear and branching time are employed.

Comparison of different classifications allows one to distinguish three types of models in the *substantiality dimension*: state based, action based, and event based models. The dichotomy “intentionality/extensionality” from [8] reflects interpretation modality of states, actions, and events.

It is also reasonable to add one more class to the temporal dimension: models with cyclic time. Finite automata are computing models with cyclic time. Thus, in the temporal dimension, there are models with *linear*, *branching*, and *cyclic* time.

In addition, it is possible to classify models of distributed and concurrent computations (as other mathematical models) by those mathematical structures that are utilized. Thus we have: 1) automata models: Petri nets, labeled transition systems, neural networks, multihead Turing machines, cellular automata, systolic arrays, and grid automata; 2) linguistic models: formal grammars of different kinds; 3) logical models, such as logical calculi and logical varieties; 4) category models (cf., [15]); and so on.

Concurrency can be also studied on different levels:

- System level representing concurrent systems.
- Algorithmic level representing concurrent algorithms/programs.
- Process level representing concurrent processes.

Here we consider two first levels and use for the study of concurrent and distributed processes such model as grid automaton [3, 6]. This model encompasses all other existing automaton models of distributed and concurrent computations, extending their possibilities. In comparison with cellular automata, a grid automaton can contain different kinds of automata as its nodes. For example, finite automata, Turing machines and inductive Turing

machines can belong to one and the same grid. In comparison with systolic arrays, connections between different nodes in a grid automaton can be arbitrary like connections in neural networks. In comparison with neural networks and Petri nets, a grid automaton contains, as its nodes, more powerful machines than finite automata. Consequently, neural networks, cellular automata, systolic arrays, and Petri nets are special kinds of grid automata. An important property of grid automata is a possibility to realize hierarchical structures, that is, a node can be also a grid automaton. In grid automata, interaction and communication becomes as important as computation. This peculiarity results in a variety of types of automata, their functioning modes, and space organization.

Informally, such a grid automaton consists of (partially) connected and interacting computing systems. These systems can be computers in a local or global network, a net of imbedded devices, components of a single computer and so on. Although different systems in a network can function independently, their interaction is usually subjected to definite rules. These rules often form algorithms of interaction for distributed systems and devices. The main goal is to understand how control algorithms in distributed systems organize computational processes in a combined algorithmic operation. In particular, we are interested in the algorithmic level of complexity, which is defined by the class of algorithms that control processes.

## 2 Grid Arrays and Automata

We consider concurrent computations going in some system of algorithmic computing devices. It is possible to represent such a system by a grid array and model it by a grid automaton [4, 5].

**Definition 2.1.** A *grid array* is a system of information processing systems (computers, networks, imbedded systems, etc.), which are situated in a grid, called nodes, are optionally connected and interact with one another.

The concept of *grid array* is very general, allowing one to include into the grid continuous and hybrid systems, people and even non-living (natural and artificial) systems (e.g., planes, plants, atmosphere, and the Sun) when they are considered as information processing systems. One grid array can be a node of another grid array. However, the most direct application of the grid approach is to computers and their conglomerates.

Grid arrays, as a rule, have distinct layers:

1. *Hardware layer* consists of physical elements, components and/or devices, which may be considered with or without their software;
2. *Software layer* consists of program elements (instructions, operators, etc.), components, separate programs, and systems of programs together with those devices that these components and programs utilize in their functioning;
3. *Infware layer* consists of data elements (bits, symbols, etc.), groups of data elements, data bytes, and program data together with those programs that process these data and those devices that these program utilize in their functioning.

The program data may be given to the program as one portion. This is a classical model of computation such, for example, as a Turing machine. The program data may be given to the program as several portions. There are three modes of such data supply:

- The *atemporal mode* when portions of data are given in a sequence without any relation to time. This is also a classical model of computation such, for example, as a finite automaton.
- The *deterministic temporal mode* when portions of data are given at predetermined intervals/moments of time. An example of such model is given by timed automata (cf., for example, [1]).
- The *nondeterministic or random temporal mode* when portions of data are given at random intervals/moments of time. An example of such model is given by interactive automata, such as persistent Turing machines [9] or global Turing machines [1].

Grid arrays are modeled by grid automata. Informally, a grid automaton is a system of automata, which are situated in a grid, optionally connected, and interact with one another through a system of connections/links.

The basic idea of interacting devices and communicating automata and processes is for a transmitting system/process to send a message to a port and for receiving system/process to get the message from a port. Thus, to formalize this structure, we assume, as it is often true in reality, that connections are attached to automata by means of ports. Ports are specific automaton elements through which information/data come into (*output ports* or

outlets) and send outside the automaton (*input ports* or *inlets*). Thus, any system  $P$  of ports is the union of its two nonintersecting subsets  $P = P_{in} \cup P_{out}$  where  $P_{in}$  consists of all inlets from  $P$  and  $P_{out}$  consists of all outlets from  $P$ . If there are ports that are both inlets and outlets, we combine such ports from couples of an input port and an output port. There are different other types of ports. For example, contemporary computers have parallel and serial ports. Ports can have inner structure, but in the first approximation, it is possible to consider them as elementary units.

We also assume that each connection is directed, i.e., it has the beginning and end. It is possible to build bidirectional connections from directed connections.

**Definition 2.1.** A *grid automaton*  $G$  is the following system that consists of three sets and three mappings

$$G = (A_G, P_G, C_G, p_{IG}, c_G, p_{EG})$$

Here:

The set  $A_G$  is the set of all automata from  $G$ ;

the set  $C_G$  is the set of all connections/links from  $G$ ;

the set  $P_G = P_{IG} \cup P_{EG}$  (with  $P_{IG} \cap P_{EG} = \emptyset$ ) is the set of all ports of  $G$ ,  $P_{IG}$  is the set of all ports (called *internal ports*) of the automata from  $A_G$ , and  $P_{EG}$  is the set of *external ports* of  $G$ , which are used for interaction of  $G$  with different external systems;

$p_{IG}: P_{IG} \rightarrow A_G$  is a total function, called the *internal port assignment function*, that assigns ports to automata;

$c_G: C_G \rightarrow (P_{IGout} \times P_{IGin}) \cup P_{IGin} \cup P_{IGout}$  is a (eventually, partial) function, called the *port-link adjacency function*, that assigns connections to ports;

and

$p_{EG}: P_{EG} \rightarrow A_G \cup P_{IG} \cup C_G$  is a function, called the *external port assignment function*, that assigns ports to different elements from  $G$ .

If  $l$  is a link that belongs to the set  $C_G$  and  $c_G(l)$  belongs to  $P_{Gin} \times P_{Gout}$ , i.e.,  $c_G(l) = (p_1, p_2)$ , it means that the beginning of  $l$  is attached to  $p_1$ , while the end of  $l$  is attached to  $p_2$ . Such link is called *closed*. If  $l$  is a link from  $C_G$  and  $c_G(l)$  belongs to  $P_{Gin}$  (or  $P_{Gout}$ ), i.e.,  $c_G(l) = p_1 \in P_{Gin}$  (correspondingly,  $c_A(l) = p_2 \in P_{Gout}$ ), it means that the beginning of  $l$  is attached to  $p_1$  (correspondingly, the end of  $l$  is attached to  $p_2$ ). Such links is called *open*.

The automata from  $A_G$  are also called *nodes of  $G$* , and connections/links from  $C_G$  are also called *edges of  $G$* . Like ports, nodes and edges can be of different types. For instance, nodes in a grid automaton can be finite automata, Turing machines, port automata [2], vector machines, array machines, random access machines (RAM), inductive Turing machines [3], etc. Even more, some of the nodes can be also grid automata.

As a result, elements from the set  $A_G$  have they inner structure. Besides, elements from the sets  $P_G$  and  $C_A$  can also have they inner structure. For example, a link or a port can be an automaton. If we consider Internet as a grid automaton with computers as nodes, then links include modems, servers, routers, and eventually some other devices. A network adapter is an example of a port with inner structure.

**Remark 2.1.** To have meaningful assignments of ports, the port assignment functions  $p_{IG}$  and  $p_{EG}$  have to satisfy some additional conditions. For instance, it is necessary not to assign (attach) input ports of the automaton  $G$  to the end (output side) of any link in  $G$ . In the case of a neural network as a node of  $G$ , inner ports of  $G$  are assigned to this network are usually connected to open links going to (inlets) and from (outlets) neurons. At the same time, it is possible to have such ports connected to neurons directly, as well as free ports that are not connected to any element of the network. Free ports might be useful for increasing reliability of the network connections to the environment. When some port fails, it would be possible by dynamically changing the assignment function to change the damaged port by a free port.

Taking the nervous system of a human being and representing it as a grid automaton with neurons as its nodes, it natural to consider dendrites and axons as links – dendrites are input (incoming) links and axons are output (outgoing) links. Then synaptic membranes are ports of this automaton: presynaptic membranes are outlets and postsynaptic membranes are inlets. Presynaptic membranes are axon terminals, i.e., output ports are adjusted only to output links, while postsynaptic membranes are parts of dendrites and bodies of neurons, i.e., input ports are adjusted both to nodes (automata) and to input links. Cell membranes in general and neuron membranes, in particular, give examples of ports with a complex inner structure.

**Remark 2.2.** Representation of grid automata without ports is the first approximation to a general network

model [4, 5], while representation of grid automata with ports is the second (more exact) approximation. In some cases, it is sufficient to use grid automata without ports, while in other situations to build an adequate, flexible and efficient model, we need automata with ports as nodes of a grid automaton.

Thus, the difference is that a grid array consists of real/physical information processing systems and connections between them, while a grid automaton consists of abstract automata as its nodes. Nodes in a grid automaton can be finite automata, Turing machines, vector machines, array machines, random access machines, inductive Turing machines, etc. Even more, some of the nodes can be also grid automata.

**Remark 2.3.** Grid automata form a base for the development of a mathematical schema theory, in which a schema is informally a grid automaton with variables. This model of a schema encompasses those schemas that are utilized in programming, mathematics, logic, brain theory, neurophysiology, and psychology.

A grid automaton is realized (situated) on a grid.

**Definition 2.3.** The *grid*  $G(A)$  of a grid automaton  $A$  is the generalized oriented multigraph that has exactly the same vertices and edges as  $A$ , while its adjacency function  $c_{G(A)}$  is a composition of functions  $p_{IA}$  and  $c_A$ , namely,  $c_{G(A)}(l) = p_{IA}^*(c_A(l))$  where  $l$  is an arbitrary link from  $C_A$ ,  $A'_A$  and  $A''_A$  are disjoint copies of  $A_A$ , and  $p_{IA}^* = (p_{IA} \times p_{IA}) * p_{IA} * p_{IA} : (P_{IAin} \times P_{IAout}) \cup P_{IAin} \cup P_{IAout} \rightarrow (A_A \times A_A) \cup A'_A \cup A''_A$ . Here  $\times$  is the product and  $*$  is the coproduct of mappings in the sense of category theory [11].

A grid automaton  $B$  is described by three grid characteristics, three node characteristics, and three edge characteristics.

**Grid characteristics** are: the *space organization* or *structure* of  $B$ ; the *topology* of  $B$  is determined by the type of the node neighborhood; and the *dynamics* of  $B$  determines by what rules its nodes exchange information with each other and with the environment of  $B$ . Topology of computer networks gives an example of the grid automaton topology [11].

**Node characteristics** are: the *structure* of the node, including structures of its ports; the *external dynamics* of the node determines interactions of this node; and the *internal dynamics* of the node determines what processes go inside this node.

**Edge characteristics** are: the *external structure* of the edge; properties and the *internal structure* of the edge; and the *dynamics* of the edge determines edge functioning.

Grid automata can give many representations of one grid array. There are three main types of such representations of a grid array  $G$  by a grid automaton  $A$ :

*Hardware representation* corresponds nodes of the automaton  $A$  to the devices and/or their groups in the array  $G$ , where a device may be a separate computer with all its software and infware (for example, databases) or the whole network of computers, servers and other computer systems with their resources.

*Software representation* corresponds nodes of the automaton  $A$  to the programs and/or their systems in the array  $G$ , where programs are taken with their resources: devices and data that are used by a program.

*Infware representation* corresponds nodes of the automaton  $A$  to the services that are provided by the array  $G$  and/or their systems, where services are taken with their resources: devices, programs, and data that are used by a service.

There are three main functioning modes of grid arrays and automata [6].

**Definition 2.3.** The *synchronous mode* when all nodes/automata make each step of their computation at the same time.

**Definition 2.4.** The *synchronized mode* when there is a sequence  $ST$  of temporal points such that at each point all nodes/automata finish some step of computation and/or begin the next step.  $ST$  is called the synchronization sequence for the process.

**Definition 2.5.** The *asynchronous mode* when different nodes/automata function in their own time.

The synchronized mode allows and the asynchronous mode implies branching and cobranching time in the grid array. According to the system theory of time, branching means that time becomes independent in some subprocesses, while cobranching means that independent times in some subprocesses become synchronized, i.e., reduced to time in a superprocess.

### 3 Algorithmic Organization of Grid Arrays and Automata

Several concurrent processes can go on even in one computing device (information processing system with one processor) when different programs realize these processes. Moreover, even one sufficiently complex program can organize many processes. Operating system of a computer is an example of such a program.

It is possible to divide all concurrent computations into three types: free concurrent computations, partially free concurrent computations, and algorithmic or procedural concurrent computations.

In turn, there are two types of procedural (algorithmic) concurrent computations: implicitly procedural (algorithmic) concurrent computations and explicitly procedural (algorithmic) concurrent computations.

**Definition 3.1.** Concurrent computation is called *free* if interactions between processes go without any rules.

For instance, it is demonstrated in [4] that a system of two finite automata interacting without any rules can eventually compute any function. However, when interaction of processes is not specified, at least, by some rules, the enveloping (computational) process can lead to deadlocks, data corruption when different processes change common data without concordance, and other safety violations.

**Definition 3.2.** Concurrent computation (functioning of a grid array/automaton) is called *partially free* if not all interactions between processes are specified by rules.

**Definition 3.3.** Concurrent computation (functioning of a grid array/automaton) is called *implicitly procedural (algorithmic)* if all interactions between processes go according to local rules (where each set of local rules form an algorithm of local interactions).

For instance, each process (algorithm or device) in a system has its own interaction rules. However, for some processes these rules can coincide.

**Definition 3.4.** Concurrent computation (functioning of a grid array/automaton) is called *explicitly procedural (algorithmic)* if all interactions between processes go according to some system of rules (algorithm).

Here we consider explicitly algorithmic concurrent computations.

**Definition 3.5.** Explicitly algorithmic functioning of a grid array/automaton is called *algorithmic operation (AO)*.

Formally it is possible to represent such computations as algorithmic operations on algorithms because, as we assume, all processes are realized by computing devices and it is a general presupposition that computing devices function under control of algorithms. In general, an operation on algorithms (AO) is a mapping of systems of algorithms into a procedure/algorithm [3]. Thus, if we represent concurrent processes  $P_1, \dots, P_n$  by their control algorithms  $A_1, \dots, A_n$  and assume interaction of processes is organized by one algorithm  $A$ , then the procedure that organizes the whole process composed of  $P_1, \dots, P_n$  is denoted by  $A(A_1, \dots, A_n)$ . According to permitted interactions, algorithms  $A_1, \dots, A_n$  form a network (grid). This network is a kind of a grid array. Taking abstract automata realizing/representing each of these algorithms, we obtain a grid automaton.

**Theorem 3.1.** If the grid array  $W$  that consists of algorithms  $A_1, \dots, A_n$  functions in a synchronous mode, then the implicitly algorithmic concurrent computation realized by the array  $W$  is equivalent to an explicitly algorithmic concurrent computation.

Although we denote the resulting algorithm by  $A(A_1, \dots, A_n)$ , different types of interactions demand for their modeling different kinds of algorithmic operations.

Let us consider two main types of such operations: *endstate* and *dynamic algorithmic operations*.

Given algorithms  $A, A_1, \dots, A_n$ , the *endstate algorithmic operation* (EAO) induced by an algorithm  $A$  gives a new algorithm  $A_e(A_1, \dots, A_n)$ , which is defined for input data  $x_1, \dots, x_n$  in the following way:

$A_e(A_1, \dots, A_n)(x_1, \dots, x_n)$  is equal to  $A_e(A_1(x_1), \dots, A_n(x_n))$  when all  $A_1, \dots, A_n$  give results  $A_1(x_1), \dots, A_n(x_n)$  for corresponding data  $x_1, \dots, x_n$ , and is undefined (we denote this by writing  $= *$ ) when at least one  $A_i(x_i)$  is undefined, what is denoted by  $A_i(x_i) = *$ .

Informally, algorithms  $A_1, \dots, A_n$  work separately until all of them produce results, then the algorithm  $A$  takes these results as input data and works with them.

Let us consider how types of algorithms influence the type of the algorithmic operation. Recursive algorithms, i.e., algorithms that are equivalent to Turing machines [6], are an important type of algorithms.

**Theorem 3.2** [3]. If all algorithms  $A, A_1, \dots, A_n$  are recursive, then  $A_e(A_1, \dots, A_n)$  is a recursive algorithm.

Inductive algorithms, which are equivalent to inductive Turing machines [6] form another important type.

**Theorem 3.3.** If the algorithm  $A$  are recursive and all algorithms  $A_1, \dots, A_n$  are inductive of the order less than  $k$ , then  $A_e(A_1, \dots, A_n)$  is an inductive algorithm of the order less than  $k$ .

## 4 Dynamic Algorithmic Operation

*Dynamic algorithmic operations* (DAO), in contrast to endstate AO, are performed so that the algorithm  $A$  interferes into functioning of algorithms  $A_1, \dots, A_n$ . Algorithm  $A$  can: open access to common data; form, open or close connections between different nodes of the grid automaton; organize interaction (form and support a communication or interaction space in the sense of [7]; change data itself; change algorithms  $A_1, \dots, A_n$  and so on. This can be done in many ways. As a result, dynamic algorithmic operations have many types, reflecting various compositions of algorithms  $A_1, \dots, A_n$  by means of algorithm  $A$ .

**Definition 4.1.** A *synchronous dynamic algorithmic operation* (SDAO) is realized so that all  $A, A_1, \dots, A_n$  do one step at the same time. It is denoted by  $A_{\text{SDAO}}(A_1, \dots, A_n)$ .

*Synchronized dynamic algorithmic operation* (sDAO) has three main types:

- *Synchronized by time dynamic algorithmic operation* (tsDAO);
- *Synchronized by state dynamic algorithmic operation* (ssDAO) where it may be a state (states) either of a control device or of the memory or of both:

- *Synchronized by action dynamic algorithmic operation* (asDAO) if exchange (action of the control algorithm) is possible after all (some) of the interacting automata perform a definite action (an action from some set).

**Definition 4.2.** *Synchronized by time dynamic algorithmic operations* (tsDAO) are organized so that interference of  $A$  into functioning of  $A_1, \dots, A_n$  is performed only at definite points in time. It is denoted by  $A_{\text{tsDAO}}(A_1, \dots, A_n)$ .

An important class of tsDAO form *synchronized dynamic algorithmic operations with a common time unit* (cuDAO),

**Definition 4.3.** *Synchronized dynamic algorithmic operations with a common time unit* (cuDAO) are organized so that there is a common time unit  $t_0$ , each  $A_i$  performs one operation in time that is equal to some number of time units  $t_0$ , and interference of  $A$  into functioning of  $A_1, \dots, A_n$  is performed only at points from the synchronization sequence for the computational process. It is denoted by  $A_{\text{cuDAO}}(A_1, \dots, A_n)$ .

**Definition 4.4.** *Synchronized by state dynamic algorithmic operations* (ssDAO) are organized so that interference of  $A$  into functioning of  $A_1, \dots, A_n$  is performed only at definite states of devices that realize algorithms  $A_1, \dots, A_n$ . It is denoted by  $A_{\text{ssDAO}}(A_1, \dots, A_n)$ .

An important class of ssDAO form *persistent dynamic algorithmic operations*.

**Definition 4.5.** *Persistent dynamic algorithmic operation* (psDAO) are organized so that interference of  $A$  into functioning of  $A_1, \dots, A_n$  is performed only when each  $A_i$  completes one cycle of its computation.

Persistent Turing machines give an example of such operation. A *persistent Turing machine* is a Turing machine with input and output tapes that receives from time to time some input (from the environment), but does not begin to process the next input until it finishes to work with the previous input coming to a final state [9].

It is possible to consider *passive persistent dynamic algorithmic operation* (ppsDAO) when all (machines) algorithms in each action are divided into active that give information or send a message and passive that receive information/messages and the algorithm  $A$  allows passive algorithms to receive messages only when they finish a definite cycle of computation.

**Definition 4.6.** *Synchronized by action dynamic algorithmic operations* (asDAO) are organized so that interference of  $A$  into functioning of  $A_1, \dots, A_n$  is performed only when (devices that realize) algorithms  $A_1, \dots, A_n$  perform some action. It is denoted by  $A_{\text{asDAO}}(A_1, \dots, A_n)$ .

Let us find relations between different types of algorithmic operations and classes of algorithms.

**Theorem 4.1.** If all algorithms  $A, A_1, \dots, A_n$  are recursive, then  $A_{\text{SDAO}}(A_1, \dots, A_n)$  is a recursive algorithm.

**Theorem 4.2.** If the algorithm  $A$  is recursive and all algorithms  $A_1, \dots, A_n$  are inductive of the order less than  $k$ , then  $A_{\text{SDAO}}(A_1, \dots, A_n)$  is an inductive algorithm of the order less than  $k$ .

As cuDAO can be reduced to SDAO, Theorem 4.1 implies the following result.

**Theorem 4.3.** If all algorithms  $A, A_1, \dots, A_n$  are recursive, then  $A_{\text{cuDAO}}(A_1, \dots, A_n)$  is a recursive algorithm.

## 5 Conclusion

Results obtained in this paper show that under definite conditions, the algorithmic level of complexity, which is defined by the class of algorithms that control processes, does not increase in a network of algorithmic devices.

Here we study primarily concurrent processes of distributed information processing controlled by a global algorithm that takes care of all processes and devices from the grid array. However, it is interesting and important to study implicitly algorithmic concurrent processes, i.e., processes that are controlled by local algorithms. Such local algorithms can be related to devices from the grid array, programs in these devices or with processes in the grid. For instance, some algorithms of this type control access to databases, while others activate or deactivate corresponding to them programs or processes.

## 6 References

- [1] Alur, R. and Dill, D.L. (1994) A Theory of Timed Automata, *Theoretical Computer Science*, v. 126, pp. 183-235
- [2] Arbib, M.A., Steenstrup, M., and Manes, E.G. (1983) Port Automata and the Algebra of Concurrent Processes, *Journal of Computer and System Sciences*, v. 27, pp.29-50.
- [3] Burgin M. (2003) Nonlinear Phenomena in Spaces of Algorithms, *International Journal of Computer Mathematics*, v. 80, No. 12, pp. 1449-1476
- [4] Burgin M. (2003) From Neural networks to Grid Automata, in Proceedings of the IASTED International Conference "Modeling and Simulation", Palm Springs, California, pp. 307-312
- [5] Burgin M. (2003) Cluster Computers and Grid Automata, in Proceedings of the ISCA 17<sup>th</sup> International Conference "Computers and their Applications", ISCA, Honolulu, Hawaii, pp. 106-109
- [6] Burgin, M. *Superrecursive Algorithms*, Springer, New York, 2005
- [7] Burgin, M., Karplus, W., and Liu, D. (2001) The Problem of Time Scales in Computer Visualization, in "Computational Science", *Lecture Notes in Computer Science*, v. 2074, part II, pp.728-737
- [8] Cleaveland, R., et al, (1996) Strategic Directions in Concurrency Research, *ACM Computing Surveys*, v. 28, No. 4, pp. 607-625
- [9] Goldin, D. and Wegner, P. *Persistent Turing Machines*, Brown University Technical Report, 1988
- [10] Gupta, V *Chu Spaces: A Model of Concurrency*, Dissertation, Stanford University, 1994
- [11] Herrlich, H. and Strecker, G.E. *Category Theory*, Allyn and Bacon Inc., Boston, 1973
- [12] Heuring, V.P. and Jordan, H.F. *Computer Systems Design and Architecture*, Addison Wesley Logman, Inc., Menlo Park/Reading/Harlow, 1997
- [13] Sassone, V., Nielsen, M. and Winskel, G. (1996) Models for Concurrency: Towards a Classification, *Theoretical Computer Science*, v. 170, pp. 297-348
- [14] Van Leeuwen, J. and Wiedermann, J. (2001) The Turing Machine Paradigm in Contemporary Computing, in "Mathematics Unlimited: 2001 and Beyond", Springer, New York
- [15] Winskel, G. and Nielsen, M. (1995) Models for Concurrency, in *Handbook of Logic in Computer Science*, Oxford University Press, pp. 1-148