Proceedings of the 1998 Winter Simulation Conference
D.J. Medeiros, E.F. Watson, J.S. Carson and M.S. Manivannan, eds.

COMMUNICATING STRUCTURES FOR MODELING LARGE-SCALE SYSTEMS

Vadim E. Kotov

Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, California 94303, U.S.A.

ABSTRACT

Communicating Structures is a system abstraction that helps to model large-scale distributed systems, whose performance mostly depends on how well the data and messages traffic is organized. The whole variety of "traffic sensitive" communicating systems can be modeled using just a small number of basic primitives which are common for all such systems. The system components are represented simply as *nodes*. Each node has *memory* that may contain *items*. *Nets* are sets of *links* that connect the nodes. The items are generated at some nodes and move from node to node along links, with some delay. The item traffic models the message and data traffic in systems. Using uniform, systematic composition of the basic primitives, Communicating Structures are able to approximate the properties and behavior of a broad spectrum of large-scale communicating systems. *Communicating Structures Library (CSL)* is a core environment for the simulation of large communicating systems. CSL has been used to analyze the architecture of multiprocessor systems, global enterprise intranets, distributed mission-critical applications, and the World-Wide Web.

1 INTRODUCTION

Large-scale distributed systems such as global enterprise intranets, distributed mission-critical applications, distributed design/manufacturing systems, WWW, and e-commerce represent a challenge for system modeling and analysis, as their complexity is increasing, their solution space is huge, and customer requirements for QoS are growing. Combining hardware, netware, software, middleware, and applications into integrated systems results in the explosion of feasible system architectures.

To meet this challenge, we propose the *Communicating Structures* modeling methodology which views large distributed systems as *communicating systems* in which the main activities are related to the coordination of the *system traffic* consisting of messages and data moving among the system components (that may be complex systems themselves).

The modeling objectives are:

- evaluation of system performance in terms of average latencies, throughput, utilization, sensitivity to variation in the system and workload parameters;
- identification of congestions, bottlenecks, non-fairness, and unpremeditated behaviors.

There are three main assumptions on which the *Communicating Structures* methodology is based.

Assumption 1 *Traffic is the king.* This means that performance of the large-scale distributed systems mostly depends on how well the system traffic is organized. In system models we focus on the traffic and those features that influence the traffic. For example, in information systems these features are:

- system topologies (including hierarchies) ;
- data and application partition;
- data and application placement;
- queuing and scheduling;
- load balancing.

Assumption 2 *All systems are alike.* The whole variety of such "traffic sensitive" systems can be modeled using just small number of *basic system primitives* which are common for all these systems.

Assumption 3 *Any system can be approximated.* Using *systematic composition* of the basic system primitives and abstraction/refinement we are able to approximate the properties and behavior of a broad spectrum of large-scale systems.

This paper presents the *Communicating Structures Library (CSL),* a core object-oriented environment for the modeling and analysis of large communicating systems based on the *Communicating Structures* methodology. CSL has been used to analyze architectures of interconnect fabrics, multiprocessor systems, and global corporate computing environments.

The system components are represented in a*Communicating Structure* simply as *nodes*. Each node has *memory* that may contain *items*. *Nets* are sets of *links* that connect the nodes. The items are generated at some nodes and move from node to node along links, with some delay. Items may be modified by nodes. The item traffic models the data traffic in the system, which is represented as a communicating structure.

Items, nodes, memories, and nets may be elementary or may have some aggregate structure.

The CSL objects may be assigned different attributes (numbers, variables, functions, and processes) which

- define quantitative parameters such as the number of subobjects in an object, time constraints, etc.;
- locate an object in the model hierarchy such as, the object's name, its relative address in the hierarchy tree;
- change the behavior of objects;
- provide an input data for objects and register their behavior and for output and further analysis;
- provide data and functions for analytical modeling.

A model in CSL is a hierarchy of nodes with one top node that has no parent. Nodes are assigned processes that are invoked to generate items, receive/send items from/to other nodes, and to transform the items if necessary.

In the case of simulation, CSL is accumulating generic or parameterized CSL objects, functions, and processes that may be quickly assembled into a particular simulation model and tuned for a specific case study using input parameters.

CSL is a system analysis package, not a universal modeling language with the emphasis on the precise specification of systems' structure or behavior. It also supports a programming, not a pictorial, style of modeling (though it has graphical interface and visualization support for the analysis of the modeling results). The CSL hierarchy is based on C++ classes and CSL concurrency uses the main structures of C++/CSIM, a process-oriented discrete-event simulation package (Schwetman 1995). CSL is transparent in that sense that the user can use not only the CSL constructs but also everything that is below them: C++/CSIM, C++, and plain C. More details on CSL can be found in the HP Labs report (Kotov et al. 1998).

2 THE CSL STRUCTURE

There are several levels of the modeling primitives and constructs in the CSL environment.

The lowest, CSL BASE level, is formed by data structures, classes, and algorithms that allow us to comfortably define, construct, and modify the components of Communicating Structures.

The second, CSL OBJECTS level, contains the basic components of Communicating Structures and the instruments to assemble them.

The third level, CSL BASICS, provides means to build simulation or analytical models out of components of the previous level.

The next level contains the generic CSL PARTS KIT which serves to customize and refine Communicating Structures to make them adequate for the specification and evaluation of particular types of systems. It will also accumulates mathematical and statistics libraries.

Finally, CSL DOMAIN LIBRARIES are built by users and provide the means to make the construction and analysis of models in different domain-specific areas fast and reliable.

The CSL GUI provides the graphical and visualization means for the easy construction, running and analysis of the CSL models, particularly during the prototyping and debugging stages.

3 THE CSL BASE

3.1 C++/CSIM Constructs

A set of modified CSIM structures is introduced to generate and coordinate concurrent processes. A *process* is a C++ procedure that executes a *create* statement. This statement invokes a new thread that proceeds concurrently with the process that invoked it.

The mechanisms to organize the interactions between the processes are *mailboxes*, *facilities*, and *events*.

The CSL class *Mailbox* is derived from the CSIM *mailbox*, and all CSIM mailbox operations and functions are valid in CSL. Mailboxes are used for interprocess communication. A process can *send* a message which is just an integer or a pointer to a mailbox and *receive* a message from a mailbox. When a process does a receive operation on an empty mailbox, it automatically waits until a message is sent to this mailbox. The CSL Mailbox is augmented by an additional operation *send_with_delay* that makes it possible to send a message to a mailbox with some time delay.

The semantics of the CSL class *Facility_ms* is similar to that of CSIM *facility_ms*, but it is implemented via *Mailbox*. This was done in order to avoid the CSIM restrictions on the *release* operation. Facility_ms models a resource. It contains a single queue and several servers.

Only one process at a time can hold a server after executing the *reserve* statement. If there is no available server, the process waits in the queue until one of the servers is *released* and there is no process waiting in the facility queue ahead of this process.

Events are used to synchronize processes. An event is a state variable with two states, *occurred* and *not occurred*, and two queues for waiting processes. One of these queues is for processes that have executed the *wait* statement (and are in a waiting state) and another is for processes that have executed the *queue* statement (and also are in the waiting state). When the event occurs, by executing the *set* statement, all waiting processes and only one of the queued processes are allowed to proceed. The statement *clear* resets the event to the not-occurred state.

3.2 Names, Parameters, Attributes, Trees

The class *CSL_Name* provides a convenient way to construct compound names which are useful for naming hierarchical objects.

Such a compound name is, in fact, a “multistring”, a string that consists of substrings delimited by a special delimiter (the default is “.”). Each substring represents a name of a predecessor of the object in the hierarchy to which it belongs. For example, a subobject of an object may be given a “full name” which contain the name of this object as a prefix and the subobject’s name as its “first name”.

To easily parameterize the CSL models, the classes *Parameter* and *Attribute* are introduced. The external parameter makes it possible to add to an input file a named input string and then to convert it in a CSL program into a value of a simple type (integer, double, string), into a list of values, into a list of lists, etc.

The class *Attribute* provides the connection between structural attributes and external parameters through regular expression-matching between the structural name and the external parameter. Each *Attribute* has a name associated with it. It may be, for example, the name of an object or the name of another element related to the object. Each *Attribute* also has a match, which is a pointer to a *Parameter* that is the most specific matching parameter.

The classes *Observe* and *Utilize* provide external visibility for statistic and other computed values. Each observe has a name and a value, which keep track of time-valued observables. These are values such as utilization, for which the average value over time, rather than just the average of a number of observations, is critical.

4 THE CSL OBJECTS

The basic elements of *Communicating Structures* are *items*, *memory*, *nets* and *nodes*.

Nodes typically generate, receive, store, forward, and, perhaps, modify data abstractly presented as items. They store and retrieve items in the node’s memory. Nets connect the nodes into a communicating structure in which the items travel from source nodes to destination nodes. These elements are derived from the common CSL class *Object*.

A CSL object may be simple or may have a hierarchical structure and include subobjects (subnodes, subnets, submemories, subitems). Any *Object* has a *Facility_ms*, an *attribute* and a *utilize* associated with it. This makes the object a resource for which concurrent processes may compete and provides “hooks” for supplying input data to objects and collecting utilization statistics.

4.1 Item

An *Item* represents an entity that migrates in a communicating system. The item has a unique (for its life cycle) *id*. Each item carries with it a pointer to the sender-node that is its birthplace and a destination path which defines the item potential route leading to its destination (maybe just to some intermediate destination.) Not all nodes that the item will actually pass need to be listed in the path. The routes between subsequent points of the path are optional and subjects to some chosen routing algorithm. The original destination path may be also modified on the way or, after the item reaches its original destination node, it may be assigned a new destination.

If an item has subitems, then these subitems may be spawned into a set of items that are issued when the item has reached its destination and is ready to disappear.

Each item may be assigned a special *ItemTag* that represents the item type and serves to distinguish between different sorts of items. The class *Item* member functions serve to modify items and to handle its time and space attributes, for example, to mark time stamps, to change the item destination, the item length, or to change its tag.

4.2 Memory

A CSL *Memory* is an *Object* that stores *Items*. In the general case, the memory is a hierarchy of (sub)memories with the ability to store items at different levels of the hierarchy. The top memory of the hierarchy is contained in a *Node*. At the bottom of this hierarchy are “simple” memories which are just arrays of *locations* holding pointers to stored items.

The class *Memory* has members (the memory size, the current number of stored items, the number of items waiting to be stored, the last-visited submemory or location) which help to monitor and control the availability of items and storage space in the memory. As a CSL *Object*, the memory can be a resource that allows us to prevent noncontrolled nondeterministic concurrent access to it.

The functions use information about the last visited submemory or location and about the predicate which defines criteria of the selection. Then it calculates a new position taking into account the previous access position and the predicate value. That gives the possibility to create specific memory access patterns; for example, those used in FIFO or Priority Queue.

4.3 Net

The *Net* is an *Object* that makes connections between the nodes. In the general case, the net inherits a multilevel hierarchy from the class *Object*. The "top" net, that is, a net with no father-net, is a part of the *Node* definition for which it provides communication links among the node's subnodes. At the bottom of this hierarchy are *Links*, "elementary" nets, each of which connects just a pair of nodes.

Each link delivers items from a *from-node* to a *to-node* with delay which is a function of the link *bandwidth* and the transferred item length (or some other item attributes). Being derived from the class *Object*, the link is a resource with some number of servers that defines the maximal number of transfers that may occur along the link simultaneously.

The net hierarchy may be treated as a hierarchy of sets of links and their subsets. This makes possible the use of structured nets to model at an abstract level switches and interconnects, as the logic of switching is conveniently expressed in the set theory terms. Some examples of this approach will be presented in Section 6.

4.4 Node

The *Node* is the main building block of Communicating Structures. Any CSL model is the top-level node. The hierarchy of aggregate node defines the static structure (topology) of CSL models: an aggregated node is an object that represents a hierarchical graph; its subnodes are its vertices and its directed arcs are the links connecting its subnodes. Both the node and its subnodes may contain memory (the class *Node* contains *Memory* as its member).

The node's internal links (if it is an aggregated node) are clustered into a *Net*. The member *net* contains all the links that connect the node with its subnodes (both ways) and the node's subnodes among themselves. The net fully determines the node's internal structure.

The *Node*'s member functions construct the node communication structure, identify specific groups of links, find paths in the internal structure of the node, and modify its structure. When given two subnodes of a node, one of the member functions, namely *path(Node *from, Node *to)* , finds the shortest local path in the node's internal structure connecting these two subnodes.

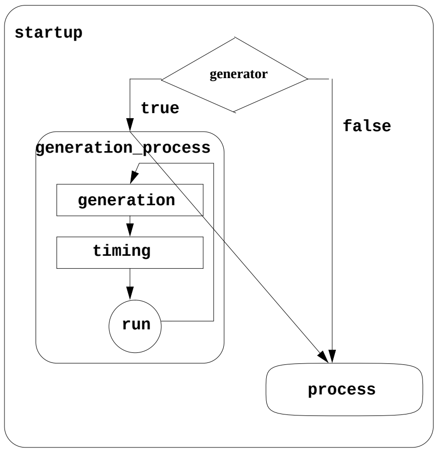

Figure 1: Startup Process

5 THE CSL BASICS

The basic CSL objects form a conceptual CSL kernel that is augmented by classes that convert the kernel into a CSL model of specific type: simulation model, or queueing model, or (stochastic) Petri Net model, etc. These classes are currently collected in the CSL BASICS sublibrary.

5.1 Process

The class *Process* introduces main generic processes that are associated with the CSL node in simulation.

The *startup* process makes the node active using the *generation_process*, which initializes item traffic from the node. It starts the main node procedure that generates the default or user-defined processes to transfer items to and from the node and allocate them in the node's memory.

Figure 1 shows the structure of the *startup* process. (In this and subsequent figures, a rectangle represents a function (procedure), a rounded rectangle represents a CSIM process, a rhombus is a condition, and a circle is loop condition.)

When a CSL model is initiated, each model's node with a special tag *generator* starts its *generation_process*. This process is recursive: it may generate an item or it may initiate another next-level generation process. Several levels of generation are useful when there is a hierarchy of generated items: messages consisting of frames, sessions consisting of messages, etc. The value of the tag *generator* specifies the number of the generation levels.

The virtual function *generation* defines the generation procedure for each level of generation. It generates an item (with the help of the virtual functions *make_item*, *destination* and *timing*) and then stores the item in the node memory with the help of the *store* function (see Figure 2).

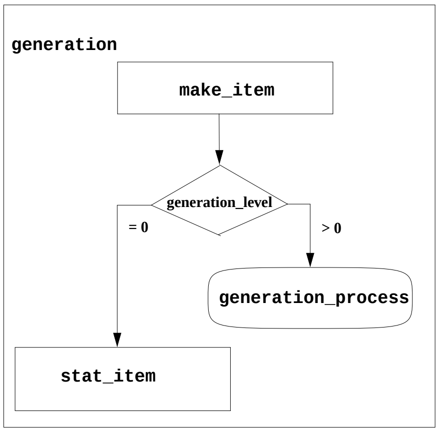

Figure 2: Generation

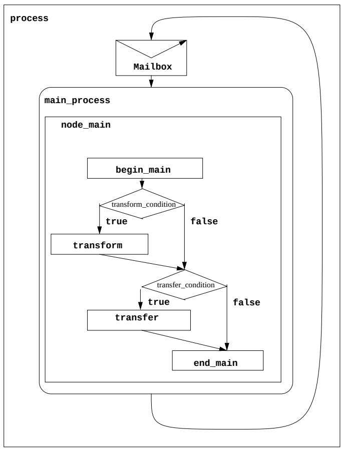

Figure 3: Main Node Process

5.2 Simulation Node

The *SimNode* represents a simulation node. It combines CSL *Node* and CSL *Process*. As the class *Process* is derived from *Mailbox*, any *SimNode* has a mailbox for communication between its processes. The node's processes generate and control item traffic and change and register the node's behavior. Most of the basic node member functions and processes are virtual and may be customized for specific purposes by the user. The default definitions of these functions provide "generic" item traffic that is generated in one subset of nodes and destined for another subset of nodes using the shortest path routing

After the generated item is stored in the node's memory, the generation process sends a message to the node's mailbox in order to activate the node's main process, which waited for a message to arrive to the mailbox. This message contains a pointer to the address of the location in which the item was stored.

The main process (see Figure 3) prescribes the node functionality and behavior. The *node_main* function executed in the process is virtual. It is defined as a superposition of several virtual functions discussed below. Hence, the *node_main* function may be either completely redefined in derived classes or it may be partially customized in only some aspects by changing the definitions of some of the constituent functions while leaving others unchanged.

The *node_main* function extracts some item from the node's memory. Which item to exract is defined either by the type of the memory (queue, priority queue, etc.) or by the user. The function analyzes the item's destination path. If the path is empty, the process completes its work without actually doing anything. Otherwise, the head of the path is studied. If it is the pointer to this node, it is deleted from the path. If the item is simple and its remaining path is empty, then the item travel in the communicating structure is terminated and the statistics related to the item is collected.

Otherwise, the *transform* function starts. This function may make some changes to the item. In particular, the function may change the item destination or make clones of this item for subsequent spawning into the communicating structure. The *transform* function is almost always customized, as it actually defines the node's functionality. The default version of *transform* is an "empty action".

After the transformation, the main process either terminates or the *transfer* function is initiated (see Figure 4). The function organizes the transfer of the item (or its subitems) to other nodes. In the default definition, it analyzes the item destination path and selects one of the possible transfer modes: monotransfer or multicast, synchronous or asynchronous.

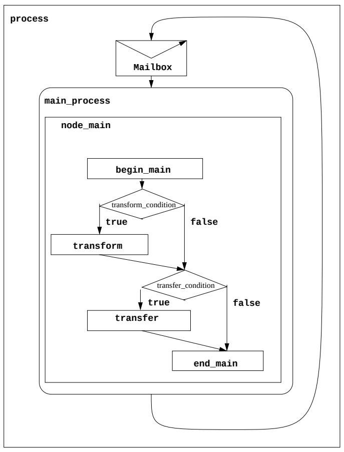

Figure 4: Transfer

5.3 Queue Node and Queue Net

The current version of CSL allow us to construct simple queueing networks (Tanner 1995). A CSL node and a CSL net (more often, a link) can be presented as a "service center" or a "queue node".

Such a queue node is modeled by a *QueueModel*, a class that constructs and executes a queueing model that is associated with the with a CSL node or a link. The type of the model is defined by the arrival time distribution, by the the service time distribution, and by the number of servers.

The input data for the queueing model are an interarrival rate and a mean service time. The model returns the the average waiting time, the average time spent in a queue node, the average number of items in the node, the average number of waiting items, and their standard deviations.

The class *QueueNet* makes it possible to describe a Jackson network of queues. In the default version of the *QueueNet*, each CSL node and each CSL link is assigned a queueing model, which is introduced using the class *QueueModel*.

Given the number of the network queue nodes and, for each queue node, the arrival rate from outside the network, the probability that an item goes from this node to another given node, the service time, and the number of servers, the *QueueNet* returns for each node: the average time spent in a queue node, the average number of items in the node, the average number of waiting items, and their standard deviations.

6 THE CSL PARTS KIT, CSL DOMAINS AND VISUALIZATION

6.1 Parts Kit

The parts kit contains sublibraries that accumulate those system templates (structures, classes, functions, and processes) that are frequently used. These templates are generic, that is, they are used quite often for various type of systems but are not basic CSL objects or functions.

For example, some specific types of memories that are derived from the class *Memory* are introduced in the "Memories" part of the KIT. Such classes as *FIFO*, *Stack* and *PriorityQueue* often serve as "control memories" that help to implicitly control the traffic in communicating structures. In many cases it is convenient to have a node memory with two submemories each of which hosts a part of the traffic going through the node. For example, one submemory may take care of the ingoing traffic and another of the outgoing traffic. (In this way one can avoid deadlock situations.) To support such types of memory, the classes *DoubleBuffer* and *DoubleFIFO*, *DoublePriorityQueue* are provided.

Different interconnecting patterns are represented by specific types of nets. For example, the *Bus* net is a communicating structure abstraction of real bus-type nets. This abstraction captures the two basic properties of busses: (1) any input point is connected to any output point, and (2) only one item at a time may be transmitted. Other examples are different types of loops, rings, crossbars and other more sophisticated connections.

6.2 Systems of Servers

Some communicating structure patterns and templates may be specialized and frequently used in domain specific models. An example of such a domain specific sublibrary is the *Systems of Servers* (SoS) Library.

The main objects of the SoS Library are:

- services
- servers
- clusters of servers
- proxies
- messages
- sessions.

Examples of the problems that can be addressed in the SoS models are:

- comparison of server network (virtual) topologies
- partition of services among servers in a cluster
- partition of services among clusters
- load balancing
- caching strategies
- admission control.

6.3 Visualization

The huge analysis and design space of large-scale communicating systems requires a special instrumentation to deal with data collection, workload and test data generation, results collection and analysis, etc. Especially useful is to visualize the model behavior, the modeling results, and provide visual support of the model debugging and validation. Figure 5 shows a "hot spots" picture of a model with traffic flowing between nodes of a hierarchical network of processing centers in a distributed enterprise computing environment.

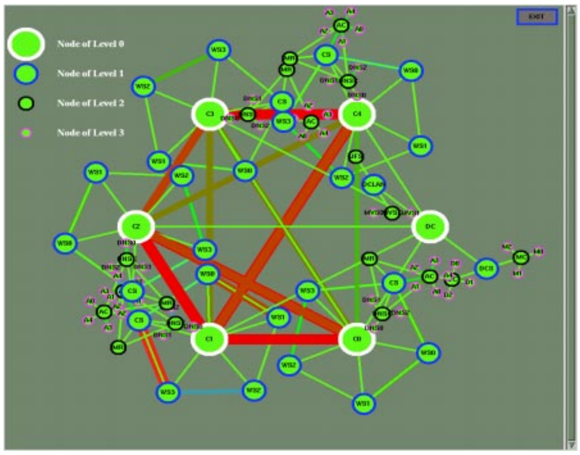

Figure 5: System Hot Spots

7 CONCLUSION

Communicating Structures Library reduces the complexity of the modeling and analysis of large-scale systems, in particular:

- simplifies construction of models of different levels of detail by using abstraction/refinement mechanisms;
- describes parallel processes and their interaction in an object-oriented way, speeding-up the model debugging and increasing the trustworthiness of models;
- speeds up the simulation of a large number of concurrent processes;
- accumulates and reuses prefabricated general-purpose and domain specific modules ("parts kit" and "domains");
- generates and analyses a larger number of the system configurations and behaviors;
- provides friendly programming and modeling infrastructure (data generation, collection, analysis, visualization, etc.).

The current version of CSL has been mostly used for the simulation of concurrent and distributed systems, because the analytical modeling methods were inapplicable to the systems under consideration. However, the analytical methods, if they work for particular types of large-scale systems, may complement simulations using the queueing analysis classes associated with the CSL nodes and a network of queues derived from the topology of a model. In a similar way, *Petri Nets*, *Colored Petri Nets*, and *Stochastic Petri Nets* can augment the CSL kernel using nodes, memories, links and items to build transitions, places, arcs, and tokens, as well as using functions and processes associated with the nodes to analyse and monitor the token traffic.

The most interesting extension of CSL is related to the intelligent browsing of the huge solution spaces for large-scale systems. The goal is not to miss good architectural solutions. This is a sort of system synthesis that relies on combining simulation, analytical and formal methods.

ACKNOWLEDGEMENTS

Lucy Cherkasova and Tom Rokicki helped to shape the basic concepts of Communicating Structures. Tom helped to implement CSL by contributing his elegant code to the CSL BASE. Lucy was the first user of the first version of CSL and the feedback from her modeling efforts drove the CSL progress at its most critical initial stage. Naftali Schwartz and Tao Zhao, interns from the New York University, contributed to the implementation of CSL structures and GUI.

The author would like also to thank Denny Georg and Rajiv Gupta for sharing their ideas, encouraging discussions, and for their overall support.

REFERENCES

Kotov,V.E., Rokicki,T.M., and Cherkasova,L.A. 1998. CSL: Communicating Structures Library for Systems Modeling and Analysis. Hewlett-Packard Laboratories Report HPL-98-118.

Schwetman, H. 1995. Object-oriented simulation modeling with C++/CSIM17. In *Proceedings of the 1995 Winter Simulation Conference*, Washington, D.C.. ed. C. Alexopoulos, K. Kang, W. Lilegdon, D. Goldsman, pp. 529 - 533, Washington, D.C.

Tanner, Mike. 1995. Practical Queueing Analysis. McGraw-Hill.

AUTHOR BIOGRAPHY

VADIM E. KOTOV is a Project Leader in the Future Systems Department at the Hewlett-Packard Laboratories in Palo Alto. Previously he worked at the Russian Academy of Sciences, heading several projects of concurrent and distributed systems. He holds a M.Sc. from the Institute of Physics Engineering in Moscow in 1963, and his Ph.D. from the USSR Academy of Sciences in 1971. His main research interest is in the theory and practice of concurrent and distributed systems with the emphasis on large-scale integrated systems. He is a correspondent member of the Russian Academy of Sciences, a full member of the Russian Academy of Natural Sciences, a member of the IFIP's Technical Committee on Foundations of Computer Science, ACM, IEEE, and is at the editorial boards of *Theoretical Computer Science* and *Parallel and Distributed Computing Practice*.