An Approach to Logic Programming of Intelligent Agents for Searching and Recognizing Information on the Internet[1]

A. A. Morozov and Yu. V. Obukhov

Institute of Radio Engineering and Electronics, Russian Academy of Sciences,
ul. Mokhovaya 11, GSP-3, Moscow, 103907 Russia
e-mail: {morozov, obukhov}@mail.cplire.ru

Abstract—The problem of designing intelligent agents for information search and recognition on the Internet is considered. A new approach to the object-oriented logic programming of Internet agents is introduced. The problem of ensuring the correct inference in conditions of permanent modification of information content on the Internet is considered. An unconventional technique for the formalization of modifiable reasoning on the Internet is described. The tools for logic programming of Internet agents based on this technique are considered.

INTRODUCTION

An ever increasing number of people are regularly searching for information on the Internet. Depending on demands of a user, this process takes different amounts of time and requires different tools for automatic data mining. For many people, information search on the Internet is still an occasional or unitary operation; i.e., a person is either not at all interested or is interested only occasionally in the subsequent modification of the information on a subject he looked for. This unitary search in the World Wide Web is automated by common retrieval systems based on linguistic methods and keyword search. It is safe to say that problems of the one-time search and recognition on the Internet can be solved by using classical recognition methods [1] (first of all, methods for structural recognition), which were developed for other applications. At the same time, as people becomes more and more dependent on information incoming from the Web (on its reliability, timeliness, completeness, etc.), the search for information on the Internet and its recognition becomes a lasting or even permanent process. In order to automate the lasting or permanent search in the Web, it is necessary to use specialized application programs (called "intelligent Internet agents"), which keep track in the information in user-defined domain of interest.

Logic programming of Internet agents is most interesting research direction in the field of information search and recognition in the Web. It has been rapidly developing during past ten years. A number of research and commercialized projects (see reviews [9, 11, 23]) proved the good prospects of this approach. It was shown that logic languages were convenient tools for analyzing complex hypertext structures in the Web owing to their high expressiveness and deductive abilities.

One can point out the following advantages of the logic approach to the programming of Internet agents.

(1) The declarative semantics makes logic languages an adequate tool for the representation of the information of a very high level of abstraction.

(2) The ideology of logic languages is based on the concept of tree search (backtrack search). This principle is very useful when searching for information in the Web.

(3) The syntax of logic languages is simpler and more expressive than that of imperative languages because (a) logic languages do not require that pointers be explicitly used and (b) the operational semantics of these languages is based on the concepts of recursion and backtracking (in contrast to imperative languages, which are based on the use of cycling and branching).

(4) Any complex data item in logic languages has a clear and unambiguous text representation.

(5) The simplicity of syntax and operational semantics of logic programs makes them a convenient object for automatic construction, analysis, and transformation.

(6) Logic languages are attractive for processing texts written in natural languages.

In this paper, we consider the methods of object-oriented logic programming of intelligent agents. These methods are based on the technique of modifiable reasoning in the dynamic Internet environment and on the Actor Prolog object-oriented logic language that we developed. In Section 1, we consider the problem of ensuring the soundness of inference in conditions of permanent changes in the information content and describe a novel technique that is used to formalize modifiable reasoning on the Internet. In Section 2, we consider the means for object-oriented logic programming of Internet agents. Section 3 is devoted to the consideration of means for logic programming of multi-

[1] This work was supported by the Russian Foundation for Basic Research, project no. 00-01-00560.

Received April 20, 2001

Pattern Recognition and Image Analysis, Vol. 11, No. 3, 2001, pp. 570–582.

agent (parallel) information search and recognition systems. In Section 4, the approach presented is compared with other approaches to the logic programming of Internet agents.

1. PROBLEM FORMULATION AND THE MAIN IDEA OF ITS SOLUTION

Consider a simplified model of a logic intelligent agent that performs the information search and recognition on the Internet.

Assume that our agent is a certain logic program that processes data extracted from the Web and output reports on gathered data (see Fig. 1). In the context of such a model, the following questions are of vital importance:

(1) Does the intelligent agent work with certain Internet resources specified in advance or are resources selected in the course of program execution and does this selection depend on the incoming data?

(2) How long should this agent operate? Certain agents are used only occasionally to carry out one-time search operations in the Web, the others uninterruptedly operate for months or even years.

Obviously, when an agent works with certain resources that are specified in advance, or when it is used for one-time search operations, the new report based on the modified data extracted from the Web can be obtained simply by running the logic program all over again. However, if the agent is to operate for a long periods of time, which are comparable with the rate of data updating in the Web, the use of standard Prolog is mathematically incorrect, because the standard strategy of Prolog execution does not ensure soundness and completeness of inference in conditions of changing source data. This problem has a purely practical aspect as well. If a logic program that was executed for a long time and independently selected necessary resources is rerun, then computational resources are spent uselessly and the gathered information is lost. More importantly, a repeated run of a logic program may fail because a part of necessary resources may be inaccessible at that very moment as often happens in the Web.

Thus, the logic programming of intelligent agents must be based on a certain inference mechanism allowing one to update the source data and to modify the statements that were inferred on the basis of these data, all other results being preserved. In other words, it is necessary to use a certain technique of modifiable reasoning. The purpose of this paper is to develop the technique that would support modifiable reasoning in the dynamic Internet environment. This technique is an alternative to nonmonotonic logic systems. The idea of this technique can be illustrated by canonical example about the ostrich Titi [2].

We write the canonical example in two different ways. The first method is based on the use of the logic program with the **not** statement, which is a certain approximated implementation of nonmonotonic logic.

```
?–can_fly (“Titi,” Answer).
can_fly (Name, “yes”):–
      bird (Name),
      not ostrich (Name).
can_fly (Name, “not”):–
      bird (Name),
      ostrich (Name).
bird (“Titi”).
ostrich (“Titi”).
```

If the database does not contain the fact formulated as *ostrich (“Titi”)*, then the proof of the assertion **not** *ostrich (“Titi”)* will succeed and Prolog will read out *Answer = “yes,”* meaning that Titi can fly. However, when the fact *ostrich (“Titi”)* is appended, this negation becomes inderivable, and Prolog will read out that Titi cannot fly.

The second method is based on the use of the technique of logic actors that we developed [17, 18, 21, 22, 24, 25]. *Logic actors* are subgoals of the logic program that can be proven repeatedly without logic program *backtracking*. In what follows, we denote logic actors by using the prefix @.

```
?–can_fly (“Titi,” Order, Answer).
can_fly (Name, Order, Answer):–
      bird (Name),
      @suitable_order (Order, Answer).
suitable_order (“conventional,” “yes”).
suitable_order (“ostriches,” “not”).
bird (“Titi”).
```

This program does the same, but the principle of its operation is different. The main distinction is that the query text (the text of a goal statement) is to be modified, not the program text. The new information is incoming in the form of terms, i.e., in the form of new values of variables in the goal statement, not in the form of logic statements.

This works as follows. In the course of logic program run, the actor *@suitable_order* is proven as a usual subgoal; Prolog will output the following result: *Answer = “yes,”* and, in addition, it will assign the value to the variable *Order*, i.e., *Order = “conventional.”* Subsequently, if it turns out that Titi is an ostrich, this information must be communicated to the

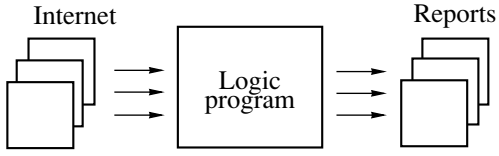

Fig. 1. A simplified model of an intelligent Internet agent.

program via the following destructive assignment operation:

Order := “ostriches.”

The soundness of inference is certainly violated by this destructive assignment. However, once the value of the variable *Order* has been changed, the logic actor mechanism will automatically restore the soundness of the inference by the repeated proving of a separate subgoal, namely, the logic actor *@suitable_order*. This time, the fact *suitable_order (“ostriches,” “no”)* will be selected, and the program will output the new result: *Answer* = “*no*.”

The second formalization method has the following merits:

(1) The new data arrive in the form of terms (data items) rather than logic statements.

(2) If the source data have been changed, it is only necessary to prove once again certain program subgoals; and

(3) Finally, the most important advantage is that we stay in the framework of classical monotonic logic with all its descriptive and deductive abilities being preserved.

To implement the logic actor mechanism, it is necessary to work out an appropriate inference strategy that supports the repetitive proving of program subgoals. It was proven in [21] that such strategies do exist; they possess the completeness (if there are no infinite loops) and soundness properties.

Based on this approach, we elaborated the Actor Prolog object-oriented programming language. In the next section, we consider the main principles of logic programming in the Actor Prolog language and give an example of a logic Internet agent.

2. OBJECT-ORIENTED LOGIC PROGRAMMING OF INTELLIGENT AGENTS

Actor Prolog is an object-oriented logic language that we designed and implemented. The purpose of designing this language was to generalize the object-oriented approach to programming, analysis and design of information systems that were based on “pure” logic means. These means provide the rigorous declarative (model–theoretic) semantics of object-oriented programs.

The problem of proper balance between logic and object-oriented programming has a long history; the literature on this subject is quite extensive (see, for example, reviews [6–8]). Studying this field, we concluded that the intuitive programmer’s concept object does not have a one-to-one counterpart in logic. When one attempts to carry over this concept to logic programming, it decomposes into at least three distinct aspects, each having the descriptive abilities, theoretical problems, and implementation facilities of their own. These are the following:

(1) *The structural aspect of the object-oriented approach.* In imperative languages, objects are natural means of program text structuring. Not only the means of program text structuring, but also the means used to control the search space correspond to this aspect in logic programming.

(2) *The dynamic aspect of the object-oriented approach.* This aspect deals with the abilities of imperative languages to express the changes in the state of objects and also with the parallel execution of separate program branches. It is this aspect that causes the greatest difficulties in the theory of object-oriented logic programming.

(3) *The informational aspect of the object-oriented approach.* It is related to problems of describing complex data structures.

In the Actor Prolog language, the descriptive capabilities in all three aspects of object-oriented approach are treated in terms of logic as logic concepts and syntactic means. For this purpose the Actor Prolog language uses the following:

1. Classes, worlds, and inheritance.

2. Repetitive proving of logic actors and parallel processes.

3. Prime and composite terms (including the so-called *underdetermined sets*).

The syntactic means that we developed are modified formulas of the first-order predicate calculus. They are logic analogs of certain means of imperative programming—classes, actors, underdetermined sets, the destructive assignment statements, and so on—but, in contrast to them, support rigorous declarative semantics of the program.

In this section, we consider two basic means of Actor Prolog, classes and logic actors. For the description of underdetermined sets, see [19] or [22], where the definition of the Actor Prolog language is given.

2.1. Classes, Worlds, and Inheritance

In Actor Prolog, the topology of the search space is controlled by the mechanism of classes.

Similar to classes in the imperative object-oriented approach, the Actor Prolog language uses syntactic constructs for program text structuring called classes. A class in a language is a set of logic clauses (facts, rules). Similar to imperative programming, a unique name is assigned to each class, and all classes are elements of certain inheritance hierarchy.

The concept “class instance” of imperative programming is also matched in the Actor Prolog language by an analog. *Class instances* (“worlds”) in Actor Prolog are particular applications of classes. Moreover, in Actor Prolog, there is no imperative statement *new* (which exists, for instance, in C++ language). Class instances in Actor Prolog are constructed implicitly as

a result of constructive proof of special formulas called *constructors.*

Class instances serve as components of the search space in the course of program execution. A class instance includes

(1) All clauses of the class and its ancestors in the inheritance hierarchy.

(2) The set of slots.

The inheritance mechanism in Actor Prolog is an analog of similar mechanisms in imperative object-oriented languages; however, it provides a somewhat different operational semantics (namely, the inheritance hierarchy defines the rules of search for program clauses). For instance, if an instance of the class *A* serves as the search space for the proof of a certain predicate *p*, then the search for the clause with the heading *p* will be performed in turn among the clauses of the class *A*, then among the clauses of the immediate ancestor of the class *A*, and so on until the appropriate clause is found or until the inheritance hierarchy is exhausted.

Thus, in Actor Prolog, the clauses that are defined in descendant classes, do not override one another, although the overriding can be modeled using nonlogic cut statement "!" In addition, Actor Prolog does not use multiple inheritance. A class can have only one immediate ancestor, because otherwise, the order of searching through the clauses of parent classes in the course of inferring is not clear.

A *slot* is a variable that is available in all clauses of a class instance. Slot names are called *class attributes*. Attributes must be declared in all classes in which the corresponding slots are used. In attribute declarations, *initializers of slots* may be specified. Initializers are language terms, constructors, or other attributes that specify the values of slots.

Here is an example of the class definition:

```
class ADDER specializing DEVICE is
a
b     = 0 –Definition of class attributes
cl    = 0 –Slots b and cl by default
sum       –contain 0
c2
[           –class clauses
table(0, 0, F, F, 0). table(0, 1, 0, 1, 0).
table(0, 1, 1, 0, 1). table(1, 0, 0, 1, 0).
table(1, 0, 1, 0, 1). table(1, 1, F, F, 1).
goal:–
      table(a, b, c1, sum, c2).
]
```

The class ADDER that represents a complete binary adder is the immediate descendant of the class DEVICE. Attributes *a*, *b*, and *c1* denote summands of the adder and the input carry bit, attributes *sum* and *c2*, the sum and the output carry bit, respectively.

The proof of a constructor (a constructor of an instance of a certain class *C*) consists of the two following steps:

Step 1. The formation of a class instance.

(a) A corresponding search space is constructed. The search space comprises the clauses of the class *C* itself and the clauses of all its ancestors.

(b) The slots of the class instance are formed. A certain initial value is assigned to each slot whenever a corresponding initializer is specified. If a slot is initialized by a constructor, then a new world that has gone through the first step of construction (the formation phase) becomes the initial value of this slot. If a slot does not have an initializer, then an anonymous variable (denoted by the symbol "_") is taken as the initial value of this slot.

Step 2. The proof of the predicate *goal* in all the worlds that were formed at Step 1.

The constructor is successfully proven if and only if the proof of the predicate *goal* in all these worlds was successful. **Remark:** In latest versions of Actor Prolog, each automatic call of a predicate *goal* is accompanied by the declaration of a new actor; that is, the prefix @ is used when the predicate *goal* is invoked.

In the example presented above, the constructor

(ADDER, a = 1, b = 1, sum = Result)

creates a new instance of the class ADDER and assign the value *Result = 0* to this variable.

In Actor Prolog, a world in which the corresponding predicate is to be proved may be explicitly indicated in any subgoal of the clause. Such call procedures are denoted by using syntactic constructs of the form

target?p(A,B,C,…),

meaning "execute the procedure *p* in the world *target*."

In the latest versions of Actor Prolog, we have also introduced syntactic means for declaring parallel processes. We dwell on this capability of the language in the section devoted to the programming of multiagent logic search and recognition systems.

The following theorem, which establishes the mathematical correctness of the class mechanism considered above, was proven in [21].

Theorem 1. *There exist global syntactic transformations (that preserve the operational semantics of programs) which convert any program written in Actor Prolog with no actors, parallel processes, and nonlogic built-in predicates into a program written in the pure Prolog (or into a formula of the Horn subset (Horn clauses) of the first-order predicate logic.)*

Thus, classes, worlds, underdetermined sets, and other syntactic means of the Actor Prolog language, which reflect the structural and information aspects of the object-oriented approach, have a standard model–theoretic semantics.

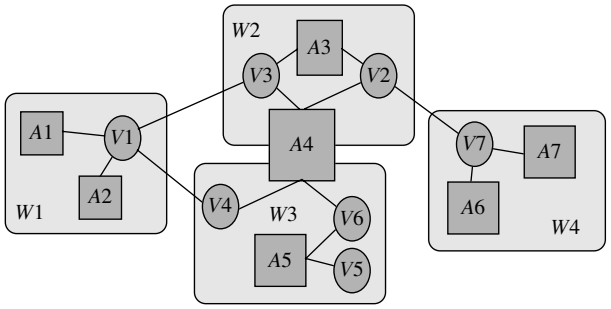

Fig. 2. Communicating logic actors.

2.2. Repeated Proving of Logic Actors

The idea of repetitively proving subgoals and the inference method implementing it (the actor mechanism) are most interesting and important elements of Actor Prolog.

A logic program in Actor Prolog is considered as a theorem that is decomposed into logic actors. *Logic actors* are repeatedly proving subgoals $A_1, \ldots, A_N$ (see Fig. 2) that communicate via common variables $V_1, \ldots, V_M$. This theorem is proven in a certain object-oriented search space made up of individual worlds $W_1, \ldots, W_K$. For instance, the rectangle A_4 in Fig. 2 denotes the actor whose subgoals are proven in the worlds W_2 and W_3.

The actor mechanism is an extension of the standard control strategy. It is based on the standard "left-to-right depth-first search with backtracking." Additional capabilities of the actor mechanism are the repetitive proof of actors and the destructive assignment of values to common variables. The repeated proving of actors is used to ensure the soundness and completeness of inference whenever the values of common variables are changed as a result of destructive assignment.

The interaction between actors is implemented by using the operation of "sound destructive assignment" that alters the values of common variables of the program. In the course of this operation, the values of common variables are changed as required and then the procedure of repeated proving of actors that depend on former values of common variables is invoked. If the repeated proving of all these actors is successfully completed, the operation is also successful; otherwise, a standard backtracking of a logic program is performed. Note that in the general case, the repeated proving of any actor can affect some other actors and cause an avalanche-like process of resolving the contradictions that have arisen in the system.

Such a coordination of actors by using the destructive assignment is performed automatically every time when each individual actor is successfully proven. In addition, this coordination procedure can be called manually by using a special built-in predicate denoted by the symbol :=. This predicate performs the destructive assignment altering the values of common variables. The declarative semantics of this predicate is exactly the same as that of conventional equality = in the pure Prolog; however, the operational semantics of this predicate is determined by the actor mechanism of the language.

Note that the repetitive proof of actors has nothing to do with the idea of so-called intelligent backtracking. Cancelling the previous results before the repeated proving of an actor is not a variety of backtracking because the previous results of proving are not removed from the stack (in contrast to the backtracking operation). Thus, the cancelled results of proving can once again be activated if a standard backtracking occurs in a program written in Actor Prolog. For this reason, the idea of repetitive proving is sometimes referred to as antibacktracking.

It should also be noted that the repeated proving of actors whose proof is not completed at the moment is forbidden in the Actor Prolog language. This rule prevents the recursive cancelling of results of proving actors, which would cause program looping.

The term actor is taken from the actor computing model due to Hewitt [3]. However, the mechanism of repetitive proving of subgoals implements another principle of communication between actors. The main distinctive features of our computing model are listed below.

(1) The information is communicated between actors only through common variables by using the destructive assignment.

(2) A logic actor is not aware which particular actors will be affected by the destructive assignment it performs. Thus, logic actors need not know one another "by name" and, therefore, the program in Actor Prolog is a strongly distributed system.

(3) The (standard) backtracking is implemented in Actor Prolog.

(4) One more interesting property of our model is the absence of spooling of data communicated between actors. A common variable may change its value several times before the repeated proving of the actor that depends on this variable is started.

(5) The drawback of our model consists in that the results of the previous proving of an actor become inaccessible once the repeated proving of this actor is started.

The detailed description of the actor mechanism can be found in the definition of Actor Prolog (see [19]). The definition of the abstract (virtual) machine of the sequential Actor Prolog is given in [21].

The declarative semantics of programs written in Actor Prolog without parallel processes is defined based on Theorem 1 as follows.

Definition 1. *A declarative (model–theoretic) semantics of a program written in Actor Prolog without parallel processes and nonlogic built-in predicates is*

the declarative semantics of the program written in pure Prolog that corresponds to the source program in the sense of Theorem 1 once all prefixes @ are removed.

In [21], the following theorems on soundness and completeness of the actor mechanism with respect to the declarative semantics are proven.

Theorem 2. (on soundness of the actor mechanism). *The control strategy of Actor Prolog with occur check, which is free from parallel processes and nonlogic built-in predicates, is sound.*

Theorem 3. (on completeness of the search space). *Any program in Actor Prolog free from parallel processes and nonlogic built-in predicates finds all existing solutions provided that no infinite loops occur in the course of its execution.*

Programs in Actor Prolog that use parallel processes, also have the model–theoretic semantics of a special kind. It will be considered in the section on multiagent logic systems. It is safe to assume that all three aspects of the object-oriented approach, which were considered above (the structural, dynamic, and informational aspects) are implemented in Actor Prolog by using purely logic means that support the model–theoretic semantics of programs. Actor Prolog, free from nonlogic built-in predicates, is a pure logic object-oriented language that supports modifiable reasoning.

2.3. An Example of the Internet Logic Agent

The program in Actor Prolog creates the set of logic actors that are linked by common variables to one another and to external information resources. If some Internet resource is altered, no repeated proving of the entire logic program is needed and only certain actors will be affected.

Consider a simple example of a logic Internet agent that performs the following task. There are many international and national government agencies that develop standards in the field of information technologies. The draft versions of standards and other electronic documents are regularly published on sites of these agencies. The goal of the agent that is considered below is to regularly check the sites of three agencies and notify the user about their modification.

```
project is ((Example2))                                          (1)
class Example2 specializing TextPage is                          (2)
node1 = (ResourceChecker,                                        (3)
        organization = "IRE RAS,"                                (4)
        location = "http://www.cplire.ru,"                       (5)
        report = self)                                           (6)
node2 = (ResourceChecker,                                        (7)
        organization = "NIST,"                                   (8)
        location = http://www.itl.nist.gov/…," (9)
        report = self)                                           (10)
node3 = (ResourceChecker,                                        (11)
        organization = "IEEE,"                                   (12)
        location=http://standards.ieee.org/…,"(13)
        report = self)                                           (14)
[ ]                                                              (15)
class ResourceChecker specializing Receptor is                   (16)
organization                                                     (17)
location                                                         (18)
report                                                           (19)
revision_period          = days(3)                               (20)
max_waiting_time         = 12.0                                  (21)
[                                                                (22)
goal:–                                                           (23)
    get_parameters(Value),                                       (24)
    write _parameters(Value).                                    (25)
write_parameters(Value):–                                        (26)
    report? write1n("Check of the Website of"),                  (27)
    report ? write1n(organization),                              (28)
    write_date_and_time(Value).                                  (29)
write_date_and_time(entry(_,Date,Time)):–!,                      (30)
    report?writeln("Recent update of the page:"),(31)
    report ? write1n(Date, "at," Time).                          (32)
write_date_and_time(_):–                                         (33)
    report ?write1n("The site is inaccessible!").                (34)
]                                                                (35)
```

The goal statement (1) of the program indicates that the execution of the program is started by the construction of an instance of the class *Example2*. The class *Example2* (2–15) is a direct descendant of the predefined class *TextPage*, which reads out data into screen windows. The instance of the class *Example2* has three slots, *node1*, *node2*, and *node3*, which contain the instances of the class *ResourceChecker*. To each instance of the class *ResourceChecker* three attributes are passed, namely, the name of a certain organization, its Internet address, and the reference to the instance of the class *Example2* itself (using the predefined slot *self*). The class *ResourceChecker* (16–35) is a descendant of the predefined class *Receptor*, which implements the program interaction with the Internet. The mode of operation of the instance of the class *Receptor* is specified by the following slots. The argument *revision_period* (20) specifies the revision of the Internet resource located at the address *location* every three days. The argument *max_waiting_time* (21) means that if no reply is received within the 12-second period in the course of the resource access, it is assumed that the resource access has failed. In each instance of the class *ResourceChecker*, the actor *goal* (23) is created.

It performs the following operations: the predicate *get_parameters* (24) returns the parameters of the specified Internet resource, and the predicate *write_parameters* (25) displays the corresponding message.

The program operates as follows. Three instances of the class *ResourceChecker* and three actors *goal* corresponding to them are created. Each of the actors is a logic statement about a certain Internet resource. This statement can be read in a declarative form in the following way: "Information about the current state of the resource *location* is displayed." In terms of logic programming, one can assert that the program execution proves the conjunction of these logic statements. Here, the principal distinction of the program in Actor Prolog from that in standard Prolog consists in the following. After a time lapse, when the state of Internet resources is changed, the logical truth of the statements of the program in conventional Prolog is violated. Whereas Actor Prolog automatically detects the changes in the resource (in the example presented above, this can be done within three days) and restores the soundness of the proof by proving once more those (and only those) actors whose resources were changed.

Thus, in contrast to standard Prolog, Actor Prolog ensures the soundness of inference in conditions of changing source data.

3. CONSTRUCTION OF MULTIAGENT LOGIC SYSTEMS

The Internet agents can be conveniently constructed of individual blocks, that is, of smaller agents (parallel processes) responsible for executing certain partial tasks. If the problem is formulated in terms of communicating asynchronous processes, then it becomes much easier to create and subsequently modify the logic agents.

We developed a logic object-oriented model of asynchronous computations for programming the multiagent logic systems and implemented this model in Actor Prolog. In this section, we consider the basic ideas that underlie this model as well as the principles of constructing multiagent systems in Actor Prolog.

3.1. Logic Object-Oriented Model of Asynchronous Computations

The principle of interacting of parallel processes implemented in Actor Prolog radically differs from the principles of parallel computing used in imperative languages. For the imperative parallel programming, the scheme that reads "Concurrently perform those operations for which the source data are ready. If the source data are not ready, do something else or hibernate" is quite natural and, perhaps, the only possible one. However, the logic programming makes it possible to view this problem in a radically different way.

The coordination principle that we developed is based on the repeated proving of subgoals. It can be formulated as follows: "Concurrently perform those operations the results of which are observable for the user. If the source data for a certain operation are not ready, inference on the basis of incomplete source data. As the new data are incoming, modify the reasoning and output refined results."

This idea is unconventional from the standpoint of classical imperative programming. Certainly, such an approach can be implemented only in the framework of a logic language, which supports the model–theoretic semantics of a program. The model–theoretic semantics of logic programs serve as a criterion for the correctness of computations; this criterion is independent of the operational semantics; it allows one to perform computations with incomplete source data.

The main advantage of our scheme is that it becomes possible not to use any methods for synchronizing parallel processes that suspend computing. This property is very useful in such fields as the visual human–machine interface and programming of decentralized distributed systems (the best example of such systems is the Internet).

We now consider in more detail the means for parallel computing in Actor Prolog.

Processes in Actor Prolog are selected class instances whose clauses are executed concurrently with clauses of other processes. Processes are denoted by enclosing the constructors of class instances in double parentheses, i.e.,

```
((ClassName, attribute1=Value1,
    attribute2=Value2,…))
```

Processes can interact through common variables in arguments of constructors and by means of predicate calls. That is why, in contrast to the classical object-oriented model where only one kind of interprocess communication is used, the computing model that we developed uses two kinds of interprocess communication, the flow and direct messages.

The difference between flow and direct messages consists in the following.

(1) Direct messages are passed directly from one process to another (in the form of predicate calls), while the flow messages are passed from one to many processes (by changing the values of common variables).

(2) Direct messages are not lost during communication, while the flow messages can cancel one another if a new (updated) value of a variable arrives before the processing of the previous value of this variable has started.

When a message is received, it is processed and the state of the process is changed. The period of process execution corresponding to the processing of a certain message, is called a phase of process execution. At each phase, the actors belonging to this process are made consistent with the information that has come from the outside. Depending on the result of repeated proving of actors, two states of a process are possible. (In this paper, we restrict ourselves to the consideration of two states of a process to simplify the presentation). Thus, it is either

(1) A proven process. This state is characterized by the consistency of all actors of the process (the proof of all actors was successful).

or

(2) A failed process. It is characterized by the fact that actors of the process are inconsistent.

Flow messages are processed in Actor Prolog in the following way. When the value of a common variable *V* that links the recipient process *P* to other processes is changed, the destructive assignment *V := NewValue* is performed in the process *P*. The result of this operation is a repeated proving of certain actors of the process *P*.

Both direct and flow messages in the model under consideration are asynchronous. In the course of execution of clauses of any process, this process can call a predicate of some other process by means of special syntactic constructs, which will be considered below. The execution of these constructs is always completed successfully and in no way affects the further execution of the process. Moreover, a specified predicate call is actually executed upon the completion of a current phase of process execution if and only if this phase is completed successfully.

The call of a predicate in some other process is denoted by using special statements of the form

Target << p(A, B, C, …)

Target <– p(A, B, C, …)

The first statement is used to denote so-called information direct messages; the second one is used to denote switching direct messages. These kinds of messages differ by the rules for processing of the received messages, namely,

(1) Once a switching message is processed, the process may become either proven or failed, while after processing an information message, the process is always proven. If the repeated proving of actors that was initiated by the received information message fails, this message is simply ignored and the process restores its former state.

(2) In contrast to switching messages, processing of information messages is suspended until the recipient process becomes proven. As long as the process is in the failed state, information messages are stored in the buffer.

Note that prior to sending direct and flow messages, all unbound variables appearing in the corresponding predicate calls and terms being communicated are replaced by the special constant # in order to prevent the chaining of unbound variables in distinct processes. This transformation preserves the soundness of a logic program with respect to the declarative semantics.

Note also that the processing of flow messages is in complete conformity with the rules for processing of switching messages presented above; therefore, flow messages can be called flow switching messages.

In our computing model, special facilities are designed to control the interprocess flow messaging. A process can declare the current value of a common variable as protected. If this is the case, then other processes that use this variable are forbidden to assign ordinary (unprotected) values to it. A protected value of a variable can only be changed for another protected value. A special syntactic designation is introduced in Actor Prolog for creating protected values of common variables; namely, the keyword **protecting** is placed before the argument name in the constructor of the process, e.g.,

((ClassName, …, **protecting** x =Value, …)).

All things considered, the logic object-oriented model of asynchronous computations can be presented in a graphic form as is shown in Fig. 3.

Each process $P_1, \ldots, P_q$ is a set of actors that are executed in certain worlds (as depicted in Fig. 2). Processes communicate through common variables $CV_1, \ldots, CV_r$, and by passing information (IDM) and switching (SDM) direct messages.

Our model is more complex in structure and has a more complex paradigm than the classical object-oriented computing model. However, in contrast to the standard object model, our model supports the declarative (model–theoretic) semantics of programs being described. In the next section, we consider the declarative semantics of parallel logic programs used in the computational model that we developed.

3.2. Declarative Semantics of Multiagent Systems

We describe and analyze multiagent logic systems by using two kinds of the model–theoretic semantics. In doing so, we proceed from the following considerations.

(1) Processes of Actor Prolog communicate strictly asynchronously, that is why, in a number of cases, it is useful and expedient to study the standard model–theoretic semantics of each process, considering these processes as independent programs.

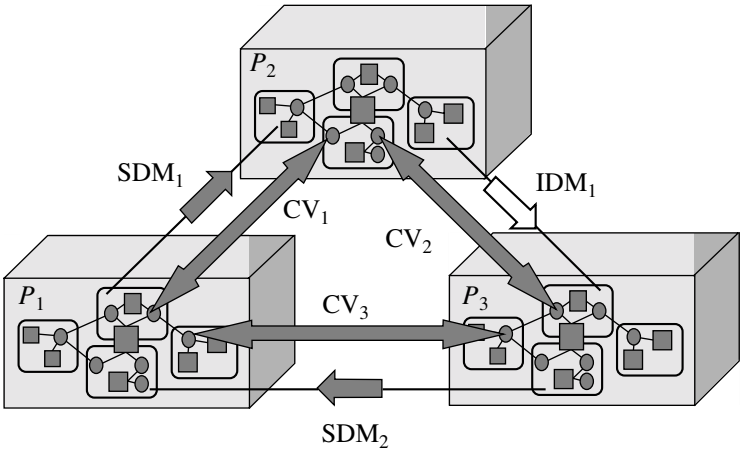

Fig. 3. Logic object-oriented model of asynchronous computations.

(2) To describe the system as a whole, we elaborated the so-called generalized model–theoretic semantics.

The generalized declarative semantics (2) is, in a sense, a less powerful description tool than the standard model–theoretic semantics (1). However, it has the following useful properties:

(a) The generalized declarative semantics is a mathematically rigorous description of the set of solutions that can be computed by a nondeterministic concurrent agent system. This description is independent of its operational semantics.

(b) The generalized declarative semantics supports a declarative programming style, which facilitates programmer's work and makes complex concurrent programs easy to read and intuitively clear.

The essence of the generalized declarative semantics consists in the following. A sequential program *P*' is constructed from the source text of a multiagent logic program *P*, which uses parallel processes and message passing, according to the following rules:

(i) All constructors of processes in the program *P* are replaced by similar constructors of class instances.

(ii) All operations of direct message passing in the text of the program are replaced by the constant "truth." As was already noted, operations of direct message passing in Actor Prolog are purely asynchronous and are always completed successfully, therefore, from the standpoint of declarative semantics, these operations can be ignored.

(iii) All nonlogic operations in the text of the program are replaced by the constant "truth."

Obviously, the program *P*' thus constructed has a standard model–theoretic semantics in the sense of Definition 1. Moreover, it is also obvious that the standard model–theoretic semantics of the program *P*' contains all values that could be computed by the source program *P*, although the operational semantics of the program *P*' differs from the operational semantics of the source program *P*.

Definition 2. *The standard model–theoretic semantics of program P' constructed according to rules (1)–(3) listed above serves as the generalized declarative (model–theoretic) semantics of the program P in Actor Prolog with parallel processes and nonlogic built-in predicates.*

Negation **not** is not used in Actor Prolog; therefore, the following proposition of soundness holds for the generalized model–theoretic semantics.

Proposition. *The control strategy of Actor Prolog with occurrence checking, parallel processes and nonlogic built-in predicates ensures the soundness of logic programs with respect to their generalized declarative semantics.*

Of course, the programs in Actor Prolog do not have a property of completeness with respect to the generalized declarative semantics. The strategy of execution of parallel logic programs that forms the basis of our logic object-oriented model of asynchronous computations is responsible for the soundness of inferene alone, allowing one to discard the completeness of inference with respect to the declarative semantics in certain cases.

3.3. Principles of Construction of Logic Systems of Communicating Agents

The logic object-oriented model of computations that we developed shapes a certain programming style and certain principles of applied system design.

(1) The use of the flow message mechanism through common variables makes individual agents (processes) of Actor Prolog sufficiently independent from one another (as was already noted, a set of communicating logic actors possesses the properties of a strongly distributed system). At the same time, links through com-

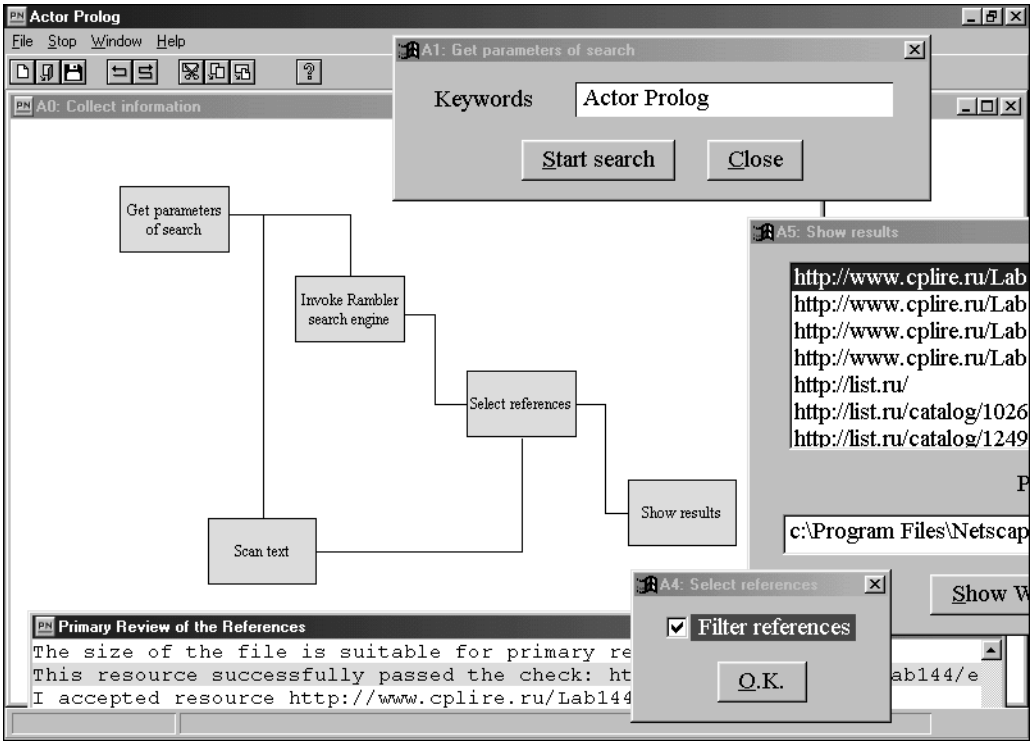

Fig. 4. An example of visual logic agent.

mon variables, in contrast to procedure calls, are static in their character and largely support the declarative style of programming. These properties of Actor Prolog facilitate and foster a wide use of component-oriented approach when designing applied systems of search and recognition. Programs written in Actor Prolog can be conveniently decomposed into separate components, parallel processes, which have meaningful declarative semantics independent from other components.

(2) Our computing model supports and facilitates the creation of visual interface of applied program. From the standpoint of logic, the human–machine interaction can be conveniently viewed as a cooperative proof of a certain theorem by an operator and a computer. In this process, the user alters the initial conditions of the problem all the time, while the computer ensures the correctness of obtained results. In Actor Prolog, the data input by a user is regarded as a destructive assignment that automatically calls the procedure of repeated proving of actors of a logic program. Thus, the actor mechanism relieves the programmer of having to "manually" describe the processing of events entering the program from the outside. This substantially simplifies and step up program design with visual user interface.

In particular, the logic object-oriented model of asynchronous computations goes well together with the ideas of visual programming using functional and data-flow diagrams.

3.4. Visual Programming of Internet Agents

We have developed experimental tools for *visual logic programming* of Internet agents. The use of SADT-diagrams [5, 20, 21] forms the basis. SADT-diagrams are a variety of functional diagrams and are widely applied for the analysis and design of complex systems [5].

The system of visual programming implements the following scheme for intelligent agent design.

(1) SADT tools are used to develop the graphic description of an intelligent agent. SADT-description is a hierarchy of blocks that receive and pass data flows (see Fig. 4).

(2) Each elementary block of a SADT-model is put into correspondence with a logic description in the form of a certain class of Actor Prolog. The source text in Actor Prolog can be written by a programmer or taken from the library of reusable modules.

(3) The graphic description of the agent is automatically translated into the text in Actor Prolog. The syntactic means of Actor Prolog make it possible to implement the block–hierarchical structure and links between blocks of a diagram in the form of communicating processes.

(4) Assembling the automatically created text and descriptions of elementary blocks, we obtain a ready-to-use program in Actor Prolog.

Experimentation with visual programming has shown that SADT-diagrams can be conveniently used not only as a visual programming language but simply as a user graphic interface. At present, visual programming system automatically creates visual interface of a logic program on the basis of source SADT-diagram (see Fig. 4).

Individual blocks of a visual user interface on the basis of SADT-diagrams are implemented by using parallel processes of Actor Prolog. As a rule, each elementary block of a diagram has its own dialog box that opens by the click of a mouse. The color of the block changes automatically depending on the state of the corresponding process. A user can interact with the blocks of the diagram in any order. Repeated alteration of any parameters entered earlier is also possible.

The tools for visual programming based on the object-oriented logic approach substantially simplify and step up the creation of Internet agents.

4. COMPARISON WITH OTHER APPROACHES TO LOGIC PROGRAMMING OF INTELLIGENT INTERNET AGENTS

Hewitt was among the first scientists who noted that the use of standard Prolog for the programming of distributed information systems was mathematically improper [4]. Unfortunately, the theoretical problem thus revealed was not timely solved and is still ignored by many researches.

Certain promising approaches to the study of declarative semantics of distributed systems were elaborated in the field of logic programming of communicating agents.

Kowalski and Sadri [12] proposed extended Prolog for the programming of systems of communicating agents. Using logic rules with the "time" parameter and a special proof procedure, they developed means for logic description of agents with memory that responded to external events. In contrast to our computing model, Kowalski and Sadri considered the interaction through the use of logic statements that described observations made by agents alone. In the framework of this approach, a declarative semantics of individual agents taken separately was considered, but the declarative semantics of the system as a whole was not supported.

Costantini proposed a promising approach making it possible to study the declarative semantics of a system of communicating agents as a whole [13]. She used special syntactic conventions and interagent communication mechanism similar to the mechanism of information direct messages used in our model of asynchronous computations.

The models of parallel computations based on various principles of interagent communication were also developed for the Internet logic programming.

Davison and Loke [10] proposed a model of parallel computations which was based on the recursive description of communicating processes (using parallel logic programming language Parlog). A distinctive feature of their model was flexible means for handling the emergencies encountered when working on the Internet. Using the developed model, Davison and Loke implemented parallel programming means in their LogicWeb logic programming system for the Internet [11].

Pontelli and Gupta [14] developed the W-ACE logic programming system also based on the recursive description of processes. In W-ACE language, the means for structuring the search space based on modal statements were introduced.

In contrast to [10, 14], Actor Prolog does not use the recursive clauses for description of interprocess communications because, in our opinion, this would complicate the task of ensuring the soundness of inference in conditions of permanent modification of the source data.

Tarau [15] developed an ingenious model of parallel computations based on "fluents," Prolog interpreters that passed the flows of found solutions to one another. This approach is close to the mechanism of "asynchronous rendezvous," which was proposed earlier by Eliëns and de Vink [16]. In our opinion, this principle of parallel process communication has good potential in the visual logic programming of Internet agents. It can be described in terms of the generalized declarative semantics we developed. At present, we experiment with the extensions of Actor Prolog of this kind.

It is the common opinion of experts in this research field that it is necessary to combine the logic approach with concepts of object-oriented programming. In our computing model and in Actor Prolog logic language, the principles of the object-oriented approach are implemented consistently and purposefully. As a consequence, they are conceptually close to and make a natural combination with the ideas of visual and component-oriented programming.

CONCLUSIONS

The approach to the object-oriented logic programming that was considered in this paper can be used to develop intelligent agents for the information search and recognition in a complex structured dynamic Internet environment. A novel technique of modifiable reasoning on the Internet forms the basis of this approach. Based on this technique, we have developed Actor Prolog object-oriented logic programming language. It ensures the soundness of logic programs (intelligent agents) functioning on the Internet environment under conditions of permanent changes of information.

The software tools that we developed make it possible to develop personalized systems (agents) for data mining on the Internet. The use of the object-oriented logic approach substantially facilitates the creation of Internet agents and their subsequent modification.

ACKNOWLEDGMENTS

We are grateful to Academician Yu. A. Zhuravlev and Professor V. A. Zakharov for fruitful discussion of the problem of describing the declarative semantics of multiagent systems.

REFERENCES

1. Gorelik, A. L., Gurevich, I. B., and Skripkin, V. A., *Sovremennoe sostoyanie problemy raspoznavaniya* (Recognition Problem: the State of the Art), Moscow: Radio i Svyaz', 1985.
2. Thayse, A., Gribomont, P., Hulin, G., Pirotte, A., Roelants, D., Snyers, D., Vauclair, M., Gochet, P., Wolper, P., Grégoire, E., and Delsarte, Ph., *Approche Logique de l'Intelligence Artificielle, vol. 2 : De la Logique Modale à la Logique des Bases de Données*, Paris: Dunod, 1989. (Translated under the title *Logicheskii podkhod k iskustvennomu intellektu: ot modal'noi logiki k logike baz dannykh* (A Logic Approach to the Artificial Intelligence: from Modal Logic to the Logic of Databases), Moscow, Mir, 1998.
3. Hewitt, C., Viewing Control Structures as Patterns of Passing Messages, *Artificial intelligence*, 1977, vol. 8, no. 3, pp. 323–364.
4. Hewitt, C., The Challenge of Open Systems, *Byte*, 1985, April, pp. 223–233.
5. Marca, D. A. and McGowan, C. L., *SADT Structured Analysis and Design Technique*, N.Y.: McGraw-Hill, 1988.
6. Alexiev, V., *Mutable Object State for Object-Oriented Logic Programming: A Survey*, Technical report TR93-15. Dep. of Computing Science, Univ. of Alberta, Alberta, Canada, 1993.
   (ftp://ftp.cs.ualberta.ca/pub/oolog/state.ps.Z)
7. Fernandes, A.A., Paton, N.W., Williams, M.H., and Bowles, A., Approaches to Deductive Object-Oriented Databases, *Information and Software Technology*, 1992, vol. 34, no. 12, pp. 787–803.
8. Davison, A.A., Survey of Logic Programming-Based Object-Oriented Languages, in: Wegner, P., Yonezawa, A., and Agha, G., Eds., *Research Directions in Concurrent Object-Oriented Programming*, MIT, 1993, pp. 42–106.
   (http://www.cs.mu.oz.au/tr_db/mu_92_03.ps.gz)
9. Davison, A., Logic Programming Languages for the Internet, in: Kakas, A., and Sadri, F., Eds., *Invited Submission for Computational Logic: from Logic Programming into the Future*, Springer, 2001.
   (http://fivedots.coe.psu.ac.th/~ad/papers/summBob.ps.gz)
10. Davison A. and Loke, S.W., *A Concurrent Logic Programming Model of the Web*, Technical Report 98/23, Department of Computer Science, Univ. of Melbourne, Melbourne, November 1998.
    (http://www.cs.mu.oz.au/~swloke/papers/conmod.ps.gz)
11. Loke, S. W., Adding Logic Programming Behavior to the World Wide Web, *Ph.D. Thesis*, Melbourne: Department of Computer Science, Univ. of Melbourne, 1998.
    (http://www.cs.mu.oz.au/~swloke/phd.ps.gz)
12. Kowalski, R. and Sadri, F., *From Logic Programming to Multiagent Systems*, Department of Computing, Imperial College, London, 1999.
    (http://www-lp.doc.ic.ac.uk/UserPages/staff/rak/papers/lpmas.pdf.gz)
13. Costantini, S., Towards Active Logic Programming, *Proc. of COCL'99 Workshop*, Paris, 1999.
    (http://costantini.dm.univaq.it/pubbls/coc199.ps)
14. Pontelli, E. and Gupta, G., *W-ACE: A Logic Language for Intelligent Internet Programming*, Department of Computer Science, New Mexico State Univ., Las Cruces, US, 1997.
    (http://www.cs.nmsu.edu/lldap/download/web.ps.Z)
15. Tarau, P., *Fluents: A Refactoring of Prolog for Uniform Reflection and Interoperation with External Objects*, Department of Computer Science, Univ. of North Texas, Denton, Texas, 2000.
    (http://www.cs.unt.edu/~tarau/research/Papers2000/fluents_cl2000.ps.gz)
16. Eliëns, A. and de Vink, E. P., Asynchronous Rendezvous in Distributed Logic Programming, in: de Bakker, J.W., de Roever, W.P., and Rozenberg, G., Eds., *Semantics: Foundations and Applications*, Springer, 1993, pp. 174–203.
17. Morozov, A. A., Actor Prolog, *Programmirovanie*, 1994, no. 5, pp. 66–78.
18. Morozov, A. A., Obukhov, Yu. V., and Oleinikov, A. Ya., Logic Programming of Open Systems, in *Logika, metodologiya, filosofiya nauki: Tezisy dokladov XI mezhdunarodnoi konferentsii*, (Logic, Methodology and Philosophy of Science. Proc. 9th Int. Conf.), Obninsk, 1995, pp. 153–156.
    (http://www.cplire.ru/Lab144/obninsk.html)
19. Morozov, A. A. and Obukhov, Yu. V., Aktornyi Prolog: *Opredelenie yazyka programmirovaniya* (Actor Prolog: Definition of Programming Language), *Preprint of Inst. of Radio Engineering and Electronics*, Russ. Acad. Sci., Moscow, 1996, no. 2(613).
    (http://www.cplire.ru/Lab144/index.html)
20. Morozov, A. A. and Obukhov, Yu. V., Semantic Analysis of Functional Diagrams of Information Systems Using Object-Oriented Logic Programming, in *Razvitie i primenenie otkrytykh sistem: Tezisy dokladov IV mezhdunarodnoi konferentsii* (Open System Development and Application. Proc. 4th Int. Conf.), Nizhni Novgorod, 1997, pp. 61–64.
    (http://www.rapros97.nnov.ru/reports/9.html).
21. Morozov, A. A., Logical Analysis of Functional Diagrams in the Interactive Design of Information Systems, *Cand. Sci. (Phys.Math.) Dissertation*, Moscow, 1998.
    (http://www.cplire.ru/Lab144/auto.html)
22. Morozov, A. A., Actor Prolog: An Object-Oriented Language with the Classical Declarative Semantics, *Proc. IDL'99 Workshop*, Paris, 1999.
    (http://www.cplire.ru/Lab144/paris.pdf)
23. Gulyaev, Yu. V., Morozov, A. A., and Obukhov, Yu. V., On the Problem of Using Logic Object-Oriented Programming in the World Wide Web, *Proc. of the Special Russian Session "The Internet Developments in Russia" (October 28, 1999). First IEEE/Popov Workshop on*

Internet Technologies and Services, Moscow, 1999, pp. 54–59.
(http://www.cplire.ru/Lab144/internet.pdf)

24. Morozov, A.A. and Obukhov, Yu.V., On the Problem of Logic Programming of Intelligent Internet Agents, in: *Diskretnye modeli v teorii upravlayushchikh sistem: Tr. IV mezhdunarodnoi konferentsii* (Discrete Models in Control Systems Theory: Proc. IV Int. Conf.), Moscow: MAKS, 2000, pp. 78–83.
25. Morozov, A.A. and Obukhov, Yu.V., On the Problem of Logical Recognition on the Dynamic Internet Environment, in: *Paspoznavanie obrazov i analiz izobrazhenii: Novye informatsionnye technologii: Tezisy dokladov V mezhdunarodnoi konferentsii* (Pattern Recognition and Image Analysis: New Information Technologies. Proc. 5th Int. Conf.), Samara, 2000, pp. 745–749. (English translation is published in *Pattern Recognit. and Image Anal.*, 2001, vol. 11, no. 2, pp. 454–457.).

Aleksei A. Morozov. Born 1968. Graduated from Baumann Moscow State Technical University in 1991. Obtained his PhD (Kandidat Nauk) degree in Physics and Mathematics in 1991. Senior Researcher at the Institute of Radio Engineering and Electronics of the Russian Academy of Sciences. Current research interests: object-oriented logic programming.

Yurii V. Obukhov. Born 1950. Graduated from the Moscow Institute of Physics and Technology in 1974. Obtained his Doctoral (Doctor Nauk) degree in Physics and Mathematics in 1992. Head of the Laboratory and Deputy Director of the Institute of Radio Engineering and Electronics of the Russian Academy of Sciences. Current research interests: data visualizing, image analysis and processing, biomedical information science, and information systems. Author of more than 70 publications. Member of Editors Board of *Biomedical Radioelectronics* journal.