Logic Object-Oriented Model of Asynchronous Concurrent Computations[1]

A. A. Morozov

Institute of Radio Engineering and Electronics, Russian Academy of Sciences,
ul. Mokhovaya 11, Moscow, 125009 Russia
e-mail: morozov@mail.cplire.ru

Abstract—In this paper, we consider the model of concurrent computations developed for logic programming of Internet agents. The purpose of creating this model of computations is to ensure the mathematical strictness of searching and recognizing information on the Internet. In the developed model of computations, the interacting concurrent processes have classical model-theoretic semantics. The model of computations is based on the new principle of interaction of processes, which do not require delay for synchronization. On the basis of this model, we have developed and implemented the concurrent object-oriented logic language and also the tools for visual logic programming of Internet agents.

INTRODUCTION

Internet agents are programs that automate retrieval, recognition, extraction, and analysis of information on the Internet oriented toward the needs of an individual user (or group of users). Agents differ from the widely used Internet retrieval systems in the following:

(1) they can autonomously operate during long periods of time (days, weeks, or more) for performing the task set by the user;

(2) as any other program, once created, an agent can be used many times, whereas a query to a universal retrieval system invokes a single operation of information retrieval.

At present, no universally accepted definition of agents exists. However, programs that are autonomous, reactive (i.e., react to external stimuli), proactive (e.g., capable of planning further actions themselves), and exhibit a social behavior (if there is a system of several agents) [1, 2] are conventionally called agents.

One of the most interesting and perspective approaches to programming Internet agents is logic programming of agents [1, 3, 4]. The urgency of this approach is determined, in particular, by the fact that the ideology and principles of logic programming correspond to the problems of retrieval, recognition, and analysis of ill-structured and also hypertext information [24, 26]. However, the main advantage of logic programming is the fact that, in the framework of this approach, there exist criteria for evaluating the mathematical strictness of information processing methods, namely, the model-theoretic semantics of programs and also the notions of soundness and completeness of logic programs.

Over the past decade, a large number of methods and means of logic programming of Internet agents were developed. They are based on different modifications and extensions of the Prolog language and also on nonclassical logics (linear, modal, F-Logic, etc.) [1, 3, 4, 24]). However, up to now, no mathematical apparatus which could provide sound and complete work of logic programs (agents) in a dynamic external environment (i.e., in conditions of permanent change and augmentation of information in the Internet) was created.

To solve this problem, we have developed the logic model of concurrent computations based on the principle of repeated proving of subgoals [21–26]. In the framework of our model, the Internet agent is a system of interacting concurrent processes. The main distinction of the developed model is the fact that it presumes the existence of classical model-theoretic semantics of Internet agents functioning under conditions of information change and augmentation. For this purpose, in the framework of the model, the classical model-theoretic semantics of individual concurrent processes and of the system of interacting processes as a whole are introduced.

In the first section of the paper, we consider the principal elements of our computational model: processes, messages, and the so-called "residents." In the second section, we consider the declarative semantics and mathematical properties of concurrent programs. In the third section, we discuss the use of the developed model for the visual component-oriented programming of Internet agents. In the fourth section, our approach to programming Internet agents is compared with other approaches.

[1] This work was supported by the Russian Foundation for Basic Research, project nos. 00-01-00560 and 03-01-00256.

Received February 28, 2003

1. BASIC COMPONENTS OF THE COMPUTATIONAL MODEL

The principle of interaction of concurrent processes developed by the author and his colleagues is a generalization of the method of speculative computations applied to the field of artificial intelligence and also to the hardware implementation of computing devices. The idea of speculative computations implies that certain branches of the algorithm can be implemented in advance, prior to the moment when it becomes clear whether the obtained data are needed for further program execution. The use of this idea and the mathematical apparatus of logic programming makes it possible not to delay concurrent processes for their synchronization. Instead of delaying the processes, we use a modification of logic inference; as a result, the general scheme of interaction of concurrent processes can be represented as follows.

Each process performs computations with data available at the present moment. If some data are not yet received, the process performs computations with incomplete data. As is shown below, the developed strategy of execution of logic programs is sound with respect to their declarative semantics; therefore, any results obtained during computations are correct with respect to the declarative semantics of the program. Computations with incomplete input data can be regarded as a certain form of computing by default. Subsequently, when new or modified input data arrive in the process, the conducted computations are modified and the earlier obtained results are refined.

Modification of logic inference in our computation model is based on the principle of the repeated proving of subgoals [21–26]. The developed model is implemented in the concurrent version of the object-oriented Actor Prolog logic language [21, 26]. Therefore, to simplify the presentation in this paper, we will use the notions and syntactic notation of this language.

1.1. Processes

The process in the concurrent Actor Prolog is the class instance, whose clauses are executed concurrently relative to clauses of other processes. In the Actor Prolog, the processes are denoted by enclosing the constructor of the class instance in double parentheses:

((ClassName, attribute1=Value1, …,

attributeN=ValueN))

The transmission of information between processes in our model is carried out by means of the following mechanisms:

(1) A process can perform an asynchronous predicate call in another process.

(2) A process can pass information to other processes by changing the value of their common variable. The arguments of the corresponding constructors of class instances can be such common variables.

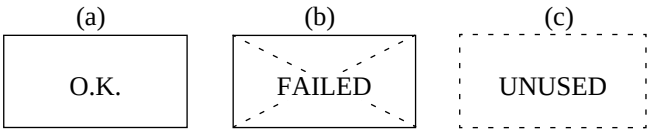

Fig. 1. States of the process. (a) Proven process, (b) failed process, (c) unused process.

(3) We have developed a special mechanism to transfer information between processes (the so-called residents) resembling the setof statement of the standard Prolog.

Thus, the processes can interact through common variables and also by means of predicate calls. That is why, in contrast to the classical object-oriented computational model using only one kind of message, two kinds of messages are distinguished in our model, direct and flow messages.

When a process handles the obtained message, its state changes. The period of process execution corresponding to the processing of a certain message is called a phase of process execution. At each phase of the process execution, the corresponding subgoals of the proof (logic actors [21–26]) are proved in accordance with the information that has come from the outside. Depending on the results of repeated proving of actors, the process is transferred into one of the following three states[2] (see Fig. 1):

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

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

(3) An unused process. The unused process can be considered as some offline component of a computing device. The processes automatically pass into the unused state and automatically recover from this state when certain special flow messages are obtained. These messages will be considered below.

The process in the unused state performs no operations, and all messages sent to it are accumulated in the buffer. Using the unused processes, the agent changes its structure in the course of execution.

1.2. 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 the predicate call), while the flow messages are passed from one to many processes (by changing the values of common variables).

[2] In the definition of the Actor Prolog [21], one more auxiliary state arising in the course of process construction is considered additionally.

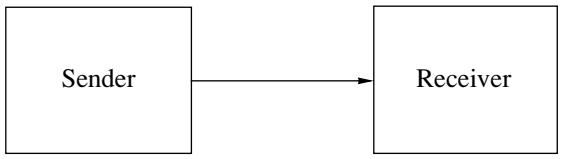

Fig. 2. Flow messages.

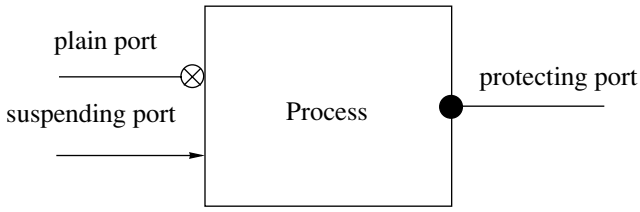

Fig. 3. Ports of processes.

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

Before the direct and flow messages are transmitted, all unbound variables in the composition of the corresponding predicate and terms are replaced by a special constant #. This prevents the chaining of variables that belong to different processes. This transformation does not violate the soundness of the logic program in respect to its declarative semantics.

Flow messages are processed in the Actor Prolog in the following way. When the value of the 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 the repetitive proving of certain actors of the process *P*. In Fig. 2, we present the graphic symbols used to denote flow messages.

Common variables that connect processes are called ports. In our computational model, special means for controlling flow message transmission are introduced. The process can declare a certain value of the common variable protected. In this case, other processes are forbidden to assign this variable the ordinary (unprotected) values. A protected value of a common variable can only be replaced by another protected value. In the Actor Prolog, the special keyword **protecting** is introduced for creating protected values of common variables. The keyword is assigned to the definitions of slots of class instances and also to the arguments of the constructors of processes, i.e.,

((ClassName,…, **protecting**: attributeN=ValueN,…)).

Note that, in the case where the process passes to the state failed or unused, the flow messages transmitted by it (including protected ones) are canceled. To do this, special empty values are sent through all the ports of the process; these values can be further replaced by any protected or unprotected values by other processes.

Another keyword, **suspending**, is used to declare the so-called suspending (switching off) ports:

((ClassName,…, **suspending**: attributeN=ValueN,…)).

Suspending ports serve for automatic passing of processes in the unused state and for their automatic recovering from this state. When a special constant # or empty value is received through the suspending port, the process is automatically passed to the unused state. When the values of all suspending ports of a process become not equal to # and to empty values, it will automatically restore its former state.

Using the suspending ports, one can create recursive definitions of processes, i.e., definitions of classes which recursively include the constructors of instances of the same class. In the general case, an attempt to create the instance of such a class leads to the construction of an infinitely large number of processes and to memory overflow. However, the use of suspending ports makes it possible to gradually construct new processes as information arrives at the suspending ports of the corresponding constructors. Thus, the Actor Prolog makes it possible to describe systems consisting of infinite number of processes and create new processes as necessary.

The ports which are neither protecting nor suspending are called plain (simple). In Fig. 3, we present graphic notation for different ports of processes.

In our computing model, direct messages are asynchronous, as also are the flow messages. In the course of execution of clauses of a process, it can call the predicate of another process (i.e., send a direct message) by using special operations of sending direct messages:

Target << predicate (A, B, …, C),

Target <– predicate (A, B, …, C).

Such an operation is always successful and does not affect the further execution of the sending process. The sending of direct messages is suspended up to the completion of the current phase of process execution and is performed if and only if this phase is completed successfully. In addition, the operations of sending direct messages are canceled in the backtracking of the logic program.

The infix << denotes the co-called informational direct messages, and the infix <– denotes the switching direct messages. These kinds of direct messages differ in the following.

(1) As a result of processing of a switching direct message, the process can become either proven or failed, while after processing of an informational direct

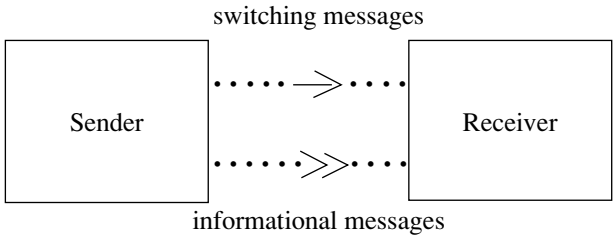

Fig. 4. Direct messages.

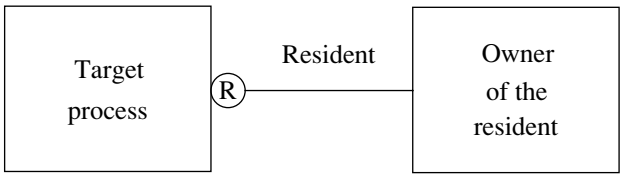

Fig. 5. Resident.

message, the process is always proven. If the execution of an informational direct message fails, this message is simply ignored and the process restores its former state.

(2) In contrast to switching direct messages, processing of informational direct messages is suspended until the recipient process becomes proven. The suspended messages are accumulated in the buffer.

In Fig. 4, we present the graphic notation used for different kinds of direct messages.

Note that the rules of processing flow messages correspond to the rules of processing switching direct messages considered above. Thus, the flow messages can be called switching flow messages.

1.3. Residents

In our computing model, the resident is a certain active entity observing the state of the assigned (target) process and sending the collected information to its owner. The owner of a resident is a certain process. In the Actor Prolog, the residents are defined via special formulas of the form

slot = target_world ?? function (A, B, …, C).

Such a formula can be assigned in the form of the argument of the constructor of the class instance or as a definition of a slot as a part of the class. In the general case, the following correspond to each resident of the program:

(1) The owner of the resident. The owner of the resident is the process that has created it.

(2) The atomic formula *function*(*A*, *B*, …, *C*). This formula denotes the call of a certain function (nondeterministic in the general case) which must be executed in the target process. The functions in the Actor Prolog are implemented with the help of the standard technique of program flattering [27].

(3) The target process of a resident, *target_world*.

(4) A certain common variable *slot*. Using this variable, the resident sends the collected information to its owner.

A resident automatically executes the assigned function in the search space of the target process. The resident creates the list of all computed values of the function and then orders this list and deletes repeated elements. After that, the resident sends the list of values of the function to its owner in the form of a protected flow message.

The resident permanently observes the state of the assigned target process. After each change in the state of the target process, the resident repeats the construction and sending of the list of values of the function. In addition, the resident can receive the information from its owner in the form of flow messages through the arguments of the corresponding atomic formula. When new values of arguments are received, the resident also performs the repetitive execution of the assigned function and sends the collected information.

Figure 5 gives a graphic representation of the residents used in our computing model.

1.4. An Example of the Graphic Description of the Agent

Let us consider an example of the graphic description of the Internet agent collecting data with the use of the *Google* and *Rambler* search engines. In Fig. 6, the functional diagram is presented. It uses the notation considered above.

Process 1 performs the interaction with a user. It inputs the list of keywords and sends to Processes 2 and 3 the switching direct message, which starts up the search process. Processes 2 and 3 interact with the corresponding search engines and accumulate the received addresses in local databases. The collected addresses are included into lists by residents and passed to Process 4 in the form of flow messages. Process 4 merges the lists of addresses and passes them to Process 5, which performs additional filtering of the received references. The necessary technique of checking Web pages is implemented in Process 6. Note that Process 6 passes itself into Process 5 via the flow message in order that Process 5 uses it as a filtering instrument. The list of checked addresses is passed to Process 7, which demonstrates the found addresses in the dialog box and makes it possible for the user to look through the found resources with the use of a standard Internet browser.

In the next section, we consider the declarative semantics of the elements of the computing model outlined above.

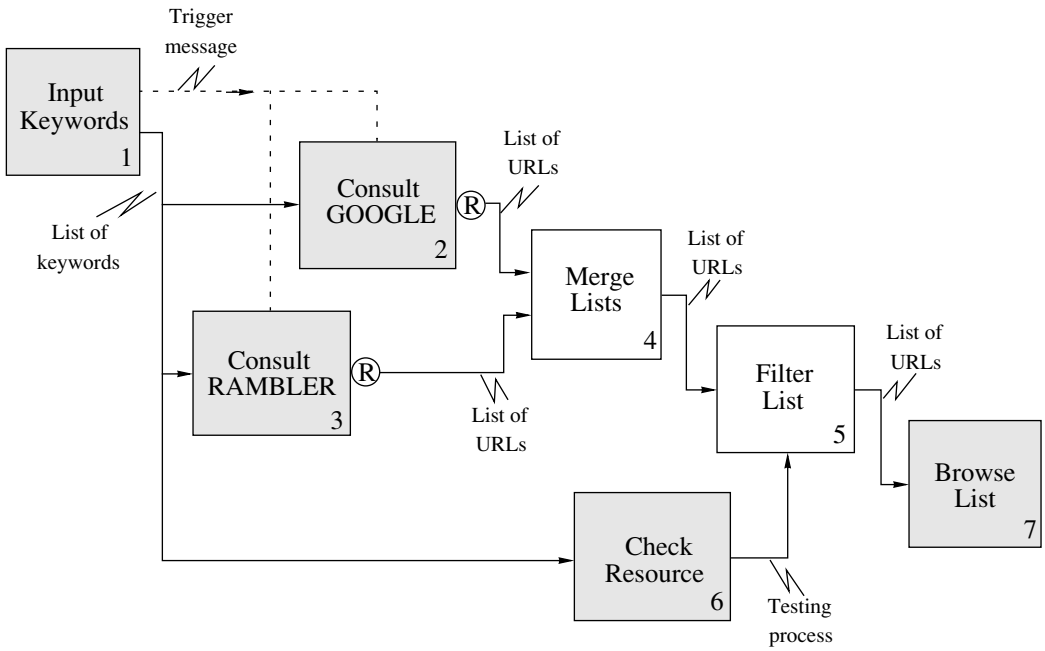

Fig. 6. Graphic description of the Internet agent.

2. MODEL-THEORETIC SEMANTICS OF CONCURRENT PROGRAMS

The model-theoretic semantics of logic programs is an important instrument evaluating the mathematical strictness of implemented algorithms. Using the model-theoretic semantics, one can evaluate the following:

(1) Soundness of algorithms. As will be shown below, the Actor Prolog ensures the soundness of programs (including concurrent ones) with respect to their model-theoretic semantics. Thus, we can be sure that the program will never compute (incorrect) values that do not belong to its model-theoretic semantics.

(2) Completeness of algorithms. Provided that certain conditions are fulfilled, we can guarantee the completeness of the logic program. This means that the program implements an exhaustive search and finds all existing solutions of the posed problem. Unfortunately, far from all logic programs are complete in respect to their declarative semantics.

An important merit of our computing model is the fact that it ensures the existence of classical (that is, based on the first-order predicate logic) model-theoretic semantics of concurrent programs.

Definition 2.1. We say that the logic program has attained its successful final state if (i) all the processes of the program are proven or unused, (ii) activation of residents is not required, and (iii) the processing of no messages by processes and residents is required.

Definition 2.2. The result computed by the program is the values of common variables connecting the processes of a program that has reached a successful final state.

Model-theoretic semantics will be defined in the following way. Below, we will define the scheme of transformations of an arbitrary concurrent logic program *P* into a sequential program *P*', for which the existence of the classical model-theoretic semantics is guaranteed. The declarative semantics of the program *P*' will be taken as the semantics of the source program *P*.

To construct the desired program *P*', we perform the following transformations:

(1) We remove from the text of the program *P* all nonlogic built-in predicates.

(2) We replace all the constructors of processes in the text of the program by usual constructors of class instances (that is, in the constructors of processes, we replace double parentheses by ordinary ones). Thereby the concurrent logic program will be transformed into a sequential program.

(3) We model the operation of suspending ports via auxiliary predicates. For example, the behavior of the process with the goal statement *goal* and two suspending ports *x* and *y* will be modeled by means of redefining the goal statement. The new goal statement *goal*' will be defined in the following way:[3]

```
goal ' :-
        x == #;
goal' :-
        y == #;
```

[3] In the Actor Prolog, the operator == corresponds to the ordinary equality = of the standard Prolog.

```
goal' :-
        ground_term(x),
        not x == #,
        ground_term(y),
        not y == #,
        goal.                    -- Old goal statement.
```

The auxiliary predicate *ground_term* is nondeterministic; it bounds the argument with all elements of the Herbran universe. This predicate is necessary in order to guarantee the correctness of using the statement **not** at the given point of computations. In the general case, this predicate outputs an infinite number of answers; therefore, this transformation can lead to infinite loop in the case where the standard control strategy of Prolog is used. However, at the moment, the only thing that matters is the presence of the declarative semantics of the program.

(4) The operation of residents will be also modeled by redefining goal statements and auxiliary predicates. For instance, if the initializer of one of the slots is a constructor of the resident of the form *slot = target ?? function*(*a, b, …, c*), we redefine the goal statement *goal* of this class as

```
goal' :-
        setof(a, b,…, c, [], Results),
        slot == Results,
        goal.                    -- Old goal statement.
```

The operation of constructing the list of solutions returned by the function *function*(*A, B, …, C*) will be implemented according to the following scheme:

```
setof(A, B,…, C, Results, Total) :-
        Result== target ? function(A, B,…, C),
        ground_term(Result),
        not is_element(Result, Results),
        setof(A, B,…, C,[Result | Results], Total);
setof(A, B,…, C, R, Total) :-
        not has_another_solution(A, B,…, C, R),
        sort (R, Total);
has_another_solution(A, B,…, C, Results) :-
        Result== target ? function(A, B,…, C),
        ground_term(Result),
        not is_element (Result, Results).
```

The auxiliary predicate *is_element* checks whether the value *Result* is contained in the list. The auxiliary predicate *sort* puts the elements of the list in order and removes the repeated elements. For sorting the elements of the list, we use the lexicographic ordering. The auxiliary predicate *ground_term* guarantees the correctness of the use of the **not** statement.

(5) We remove all operations of sending direct messages from the text of the program. As was already noted, direct messages in our model are strictly asynchronous and are always successful. Thus, from the point of view of the declarative semantics, these messages can be ignored.

(6) We remove from the text of the program the keywords **protecting**.

(7) We have obtained a sequential object-oriented program. The object-oriented constructions of the Actor Prolog can be easily modeled in the pure Prolog by simple renaming the predicates and adding certain auxiliary arguments. The detailed algorithm of such transformations can be found in [22].

As a result of transformations, we obtain the sequential program *P*' in the pure Prolog with the **not** statement. The **not** statement is not used in the Actor Prolog language. Thus, in the program *P*', the **not** statement is used only for modeling the special constructions of the language considered above. Therefore, *P*' is stratifiable and, hence, possesses the classical model-theoretic semantics.

Definition 2.3. The model-theoretic semantics of the concurrent program *P* is the model-theoretic semantics of the sequential program *P*' constructed according to rules (1)–(7).

Theorem 2.1. A concurrent logic program is sound in respect to its model-theoretic semantics.

Sketch of a proof. Considering the AND-tree of the program *P* in the course of transformations (1)–(7), one can verify that it can only decrease (at Step 1). Thus, the AND-tree of the program *P*' covers by no means less solutions than the program *P*. That is, the program *P* cannot compute the solutions not covered by the program *P*' (and, therefore, incorrect with respect to the model-theoretic semantics).

Of course, in the general case, the concurrent program *P* does not possess the completeness relative to its model-theoretic semantics. To ensure the completeness, it is necessary to impose constraints on the syntax of the program.

Theorem 2.2. The concurrent program *P* is complete in respect to its model-theoretic semantics if we have the following:

(1) There are no nonlogic built-in predicates in the text of the program.

(2) Direct messages are not used. Information between processes is transmitted only via the flow messages.

(3) The program does not get caught in an endless loop in the course of executing goal statements and functions of residents.

(4) The functions called by residents always return a finite number of values.

(5) Predicates computing data which are then sent by means of flow messages are deterministic.

(6) Information is transmitted between processes along one-direction channels only. The unidirectional data transmission in the Actor Prolog can be modeled by means of the keyword **protecting**.

(7) There is a partial ordering of processes exchanging information. That is, there is no recursive transmission of data between the processes and residents in the program.

(8) All values computed by processes and residents which must be passed to other processes and residents are ground (i.e., they do not contain unbound variables).

Sketch of the proof. There are five possible reasons for incompleteness of the concurrent program in respect to the declarative semantics. The first reason is incompleteness of computations performed inside some process. To eliminate this reason, conditions (1) and (3) are introduced. The second possible reason is the impossibility to transfer the backtracking between processes. Conditions (5) and (6) guarantee that if a process has computed and passed a certain solution into another process, then no other solution exists. Therefore, the backtracking is not necessary since it all the same would not help to find other solutions. The third possible reason for incompleteness is the replacement of unbound variables by the constant # during information transmission between processes. Condition (8) removes this reason. The fourth reason is infinite computations which may arise in the course of program execution. Such infinite loops of a program are prevented by conditions (3), (4), and (7). The fifth possible reason is the presence of several successful final states of a program. Specifically, the program can have several final states if (a) the direct messages with a generally unpredictable order of the processing are used in it; and (b) data are transmitted recursively and it is impossible to predict the order in which the processes related by common variables will be executed. To prevent these cases, we have introduced conditions (2) and (7), respectively. Therefore, provided that the enumerated conditions (1)–(8) are met, the program will be complete in respect to the model-theoretic semantics.

Thus, the concurrent computations in our model possess not only the operational but also the model-theoretic semantics. The soundness of the program is guaranteed, and, under certain conditions, the completeness of the program with respect to the model-theoretic semantics is also guaranteed. This means, in particular, that the logic language can be used for writing concurrent programs that perform an exhaustive search. This makes it possible to conclude that the developed model of concurrent computations and the concurrent object-oriented logic language implement the principles of logic programming mathematically strictly.

3. IMPLEMENTATION AND PRACTICAL USE OF MATHEMATICAL APPARATUS

In the majority of cases, the tasks of collecting and processing information which require automation with the help of intelligent agents have individual character and are formulated by concrete users. Therefore, to achieve a real economic effect from using agents, it is necessary to create technologies for programming agents which would maximally facilitate and simplify the development of such programs by final unskilled users. Ideally, the creation of the Internet agent must be as simple as the compilation of the query to the general-purpose retrieval system. The developed methods and means of logic programming create necessary prerequisites for the solution of this problem. Based on the created mathematical apparatus, the Actor Prolog, concurrent object-oriented logic language, is developed (the definition of the language including all new features can be found on the site listed in [21]). In the recent versions of the language, special means for supporting the development of the Internet agents were introduced:

(1) predefined classes for getting information according to HTTP and FTP Internet protocols;

(2) visual programming means including the translator of SADT diagrams into the concurrent Actor Prolog and the user interface management system based on the SADT diagrams;

(3) packages and other syntactic means necessary for the implementation of the visual component-oriented programming.

At present, we have implemented the operating prototype of the system of logic programming of Internet intelligent agents on the basis of the developed mathematical apparatus and the Actor Prolog language. The developed system makes it possible to create agents for the retrieval and analysis of information on the Internet. The use of the object-oriented logic approach substantially simplifies the creation of Internet agents and also their subsequent support and modification.

4. COMPARISON WITH OTHER APPROACHES

The developed method of programming the Internet agents and the model of concurrent computations embodies several ideas, each of which must be considered separately and compared with earlier approaches.

4.1. Visual Programming of Internet Agents via Functional Diagrams

The idea of visual programming of Internet agents via functional diagrams was successfully realized earlier by Mosconi and Porta [6]. They developed the *VIPERS* system [6] on the basis of a special kind of data flow diagrams. As compared with our system of visual programming of Internet agents, the *VIPERS* system possesses better graphic interface, but the blocks of diagrams are implemented in the imperative language. Thus, the *VIPERS* system does not support the declarative semantics of Internet agents and does not ensure the mathematical strictness of procedures of search and recognition of information in the dynamic Internet environment.

4.2. Concurrent Programming of Agents

The expedience of using the flow parallelism for the development of languages for programming Internet agents was also demonstrated in the *WebL* project of Kistler and Marais [7] (on the basis of combinators developed by Cardelli and Davies [8]), *Webstream* by Hong and Clark [9], *WebScript* by Zhang and Keshav [10], and information gathering plans by Barish and Knoblock [11]. The listed languages are imperative and, thus, have the same drawback as the system *VIPERS* mentioned earlier.

The logic programming languages for the Internet agents developed earlier, such as the concurrent *LogicWeb* developed by Davison and Loke [5], *W-ACE* by Pontelli and Gupta [12], *Jinni* by P. Tarau [13], OZ [14] developed under the direction of G. Smolka, *DLP* developed by Eliens and de Vink [15], *ObjVProlog-D* by Malenfant, Lapalme, and Vaucher [16], etc. [4], ensure the existence of the declarative semantics of agents but do not support the modification of logic inference in the dynamic external environment and, therefore, do not ensure the mathematical strictness of Internet agents.

4.3. Modifiable Reasoning in Multiagent Systems

As a promising approach to the logic programming of Internet agents in the dynamic external environment, the use of nonclassical (and, in particular, nonmonotonic logic systems) [3, 17, 18] is considered. We have chosen a principally different way, strictly adhering to the formalism of classical first-order logic. In our model of computations, the modification of reasoning necessary for the correct logic interpretation of changes in the external world is implemented via a special control strategy of logic language based on the principle of repeated proofs. This made it possible to mathematically strictly interpret the dynamic character of the Internet environment without loosing the expressive and deductive properties of the classical first-order predicate logic. Note that our approach is free from the drawbacks of nonmonotonic logic systems, such as absense of general validity of deduced formulas (in the classical sense), dependence of the result on the order of application of the inference rules, and the high probability of looping of logic programs.

4.4. Speculative Computations in Multiagent Systems

One of the most interesting directions in the programming of agents is speculative computations. Using this principle for the control of the process of gathering data from the Internet, Barish and Knoblock [11] achieved a significant increase in Internet agent speed. The same idea at a higher level of abstraction was studied in the work of Inoue, Kawaguchi, and Haneda [19] and also in the work of Satoh and Yamamoto [20]. It should be also noted that the idea of speculative computations is used in the *OZ* language [14] instead of backtracking.

Our model can also be considered as the implementation of the idea of speculative computations, since the synchronization of concurrent processes is not used in the Actor Prolog. This means that, in our model of computing, concurrent processes never wait for data from other processes and all computations performed by the program are in fact speculative computations. Thus, our model of computing is a mathematical basis for implementing speculative computations in multiagent systems.

CONCLUSIONS

The mathematical apparatus of logic programming of intelligent agents performing the search and recognition of information in a complex structured dynamic Internet environment is developed. The developed mathematical apparatus is based on the principle of the repetitive proving of subgoals, which makes it possible to modify the logic reasoning during and after the execution of logic programs by putting them in correspondence with the new information incoming from outside.

Based on the developed apparatus of modifiable reasoning, we have created the Actor Prolog, concurrent object-oriented logic language, which ensures the correctness of logic programs (intelligent agents) functioning under conditions of permanent change and updating of information.

The developed tools make it possible to create personal systems (agents) for gathering and analyzing information on the Internet. The use of the object-oriented logic approach substantially simplifies the creation of Internet agents, as well as their subsequent support and modification. The developed approach supports visual and component-oriented programming of Internet agents.

ACKNOWLEDGMENTS

The author is grateful to Doctor of Physics and Mathematics Yu. V. Obukhov for help and support in implementing the project, to Academician Yu.I. Zhuravlev and Professor V. A. Zakharov for fruitful discussions of the problem of describing the declarative semantics of multiagent systems, and also to Candidate of Physics and Mathematics S. V. Remizov, who provided software for debugging the prototype of the Actor Prolog.

REFERENCES

1. Sadri, F. and Toni, F., *Computational Logic and Multiagent Systems: A Roadmap*, 1999; available at http://citeseer.nj.nec.com/sadri99computational.html.
2. Huang, Z., Eliens, A., van Ballegooij, A., and de Bra, P., A Taxonomy of Web Agents, IEEE Proc. First Int. Workshop on Web Agent Systems and

Applications (WASA 2000), 2000; available at http://wasp.cs.vu.nl/wasp/papers/wasa2000.ps.

3. Eiter, T., Fink, M., Sabbatini, G., Tompits, H., Using Methods of Declarative Logic Programming for Intelligent Information Agents, *Theory and Practice of Logic Programming*, 2002, vol. 2, no. 6, pp. 645–709; available at http://arxiv.org/abs/cs.MA/0108008.
4. Davison, A., Logic Programming Languages for the Internet, in Kakas, A. and Sadri, F., Eds., *Invited Submissions for Computational Logic: From Logic Programming into the Future*, Springer, 2001; available at http://fivedots.coe.psu.ac.th/~ad/papers/summBob.ps.gt.
5. Davison, A. and Loke, S.W., A Concurrent Logic Programming Model of the Web, *Tech. Rep. 98/23*, Department of Comput. Sci., Univ. of Melbourne, Melbourne, 1998; available at http://www.cs.mu.oz.au/tr_submit/test/tr_db/mu_TR_1998_1-cp.ps.gz.
6. Mosconi, M. and Porta M., A Visual Approach to Internet Applications Development, *Proc. 8th Int. Conf. on Human–Computer Interaction (HCI'99)*, Munich, 1999, vol. 1, pp. 600–604; available at http://vision.unipv.it/research/papers/99p-webappl/webappl.pdf.
7. Kistler, T. and Marais, H., WebL—A Programming Language for the Web, Elsevier, 1998, pp. 259–270; available at http://gatekeeper.research.compaq.com/pub/DEC/SRC/technical-notes/SRC-1997-029-html/.
8. Cardelli, L. and Davies, R., Service Combinators for Web Computing, *Software Engineering*, 1999, vol. 25, no. 3, pp. 309–316; available at http://citeseer.nj.nec.com/cardelli97service.html.
9. Hong, T.W. and Clark, K.L., Concurrent Programming on the Web with Webstream, *Tech. Rep.*, Department of Computing, Imperial College, 2000; available at http://www-lp.doc.ic.ac.uk/~klc/webstream.html.
10. Zhang, Y. and Keshav, S., WebScript—A Scripting Language for the Web, Tech. Rep., Cotrnell CS Tech. Rep. TR99-1742, 1999; available at http://www.research.att.com/~yzhang/papers/webscript-tr99.pdf.
11. Barish, G. and Knoblock, C.A., Speculative Execution for Information Gathering Plans, *Proc. 6th Int. Conf. on AI Planning and Scheduling (AIPS-2002)*, Toulouse, 2002; available at http://www.isi.edu/info-agents/papers/barish02-aips.pdf.
12. Pontelli, E., Gupta G., *ACE: A Logic Language for Intelligent Internet Programming* / Department of Comput. Sci., New Mexico State Univ., Las Cruces, 1997; available at http://www.cs.nmsu.edu/lldap/download/web.ps.Z.
13. Tarau, P., Jinni: Intelligent Mobile Agent Programming at the Intersection of Java and Prolog, *Proc. of PAAM'99*, London, 1999; available at http://citeseer.nj.nec.com/tarau99jinni.html.
14. Smolka, G., The Oz Programming Model, in van Leeuwen J., Ed., *Computer Science Today*, LNCS 1000, Springer, 1995, pp. 324–343; available at http://www.mozart-oz.org/papers/abstracts/volume1000.html.
15. Eliens, A. and de Vink, E.P., Asynchronous Rendez-Vouz 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; available at http://citeseer.nj.nec.com50346.html.
16. Malenfant, J., Lapalme, G., and Vaucher, J., ObjVProlog-D: Distributed Object-Oriented Programming in Logic, *Object Oriented Systems*, 1996, vol. 3, no. 2, pp. 61–86; available at http://compscinet.dcs.kcl.ac.uk/OO/PDFPapers/oo030202.pdf.
17. Alferes, J.J. and Pereira, L.M., Logic Programming Updating: A Guided Approach, in Kakas, A. and Sadri, F., Eds., *Computational Logic: From Logic Programming into the Future*, Essays in Honor of Robert Kowalski, Springer, 2002, vol. 2, pp. 382–412; available at http://citeseer.nj.nec.com/alferes02logic.html.
18. Dix, J., Furbach, U., and Niemelae, I., Nonmonotonic Reasoning: Towards Efficient Calculi and Implementations, in Voronkov, A. and Robinson, Eds., *Handbook of Automated Reasoning*, Elsevier, 2001, vol. 2, ch. 18, pp. 1121–1234; available at http://www.cs.man.ac.uk/~jdix/Papers/01_Handbook_AR.ps.gz.
19. Inoue, K., Kawaguchi, S., and Haneda, H., Controlling Speculative Computation in Multiagent Environments, *Proc. Second Int. Workshop on Computational Logic in Multiagent Systems (CLIMA-01)*, 2001, pp. 9–18; available at http://citeseer.nj.nec.com/inoue01controlling.html.
20. Satoh, K. and Yamamoto, K., Speculative Computations with Multiagent Belief Revision, *Proc. First Int. Joint Conf. on Autonomous Agents and Multiagent Systems*, Bologna, ACM Press, 2002, pp. 897–904; available at http://agents.umbc.edu/papers/satoh.pdf.
21. Morozov, A.A. and Obukhov, Yu.V., Actor Prolog: The Definition of the Programming Language, *Preprint of Inst. of Radio Engineering and Electronics, Russ. Acad. Sci.*, Moscow, 1996 no. 2(613); the new version of the definition of the language is available at our site http://www.cplire.ru/Lab144/index.html.
22. Morozov, A.A., Logic Analysis of Functional Diagrams in the Process of Interactive Design of Information Systems, *Cand. Sci. (Phys.-Mat) Dissertation*, Moscow, 1998; available at http://www.cplire.ru/Lab144/auto.html.
23. Morozov, A.A., Actor Prolog: An Object-Oriented Language with the Classical Declarative Semantics, *Proc. IDL'99 Workshop*, Paris, 1999; available at http://www.cplire.ru/Lab144/paris.pdf.
24. Morozov, A.A., Obukhov, Yu.V., and Gulyaev, Yu.V., On The Problem of Using Logic Object-Oriented Programming in the World Wide Web, *Proc. Special Russian Session "The Internet Development in Russia", First IEEE/Popov Workshop on Internet Technol. and Services*, Moscow, 1999, pp. 54–59; available at http://www.cplire.ru/Lab144/internet.pdf.
25. Morozov, A.A. and Obukhov, Yu.V., On the Problem of Logical Recognition in the Dynamic Internet Environment, *Pattern Recognit. Image Anal.*, 2001, vol. 11, no. 2, pp. 454–457; available at http://www.cplire.ru/Lab144/pria5.pdf.

26. Morozov, A.A. and Obukhov, Yu.V., An Approach to Logic Programming of Intelligent Agents for Searching and Recognizing Information in the Internet, *Pattern Recognit. Image Anal.*, 2001, vol. 11, no. 3, pp. 570–582; available at http://www.cplire.ru/Lab144/pria570m.pdf.

27. Morozov, A.A., On Semantic Link Between Logic, Object-Oriented, Functional, and Constraint Programming, *Proc. MultiCPL Workshop*, Ithaca, USA, 2002; available at http://www.cplire.ru/Lab144/multicpl.pdf.

Aleksei A. Morozov. Born 1968. Graduated from Bauman State Technical University, Moscow, in 1991. Received his PhD (Kandidat Nauk) degree in Physics and Mathematics in 1998. Senior Researcher at the Institute of Radio Engineering and Electronics of the Russian Academy of Sciences. Scientific interests: visual and object-oriented logic programming, Internet agents.