Actor Prolog:
an Object-Oriented Language with the Classical Declarative Semantics

Alexei A. Morozov

Institute of Radio Engineering and Electronics of RAS
Moscow, Russia, RU-103907
morozov@open.cplire.ru

Abstract. This article considers the following central ideas underlying Actor Prolog: the classes and worlds, the mechanism of repetitive proof of subgoals, the underdetermined sets. The logical means of Actor Prolog cover the definitional possibilities of structural, dynamic and information aspects of the object-oriented programming. The most interesting idea of Actor Prolog is the repetitive proof of subgoals, that allows implementation of the operation of destructive assignment and thus is a new solution of the frame problem. All the logical means of Actor Prolog have a classical declarative (model-theoretic) semantics.

Introduction

Actor Prolog is a carefully designed object-oriented logic language. The purpose of the development of this language is in generalization of the object-oriented approach (OOA) in programming, analysis and designing of information systems on the basis of pure logical means with a strict declarative semantics.

What are the problems in the area of OOA, that we hope to resolve with the help of logic programming?

First, it is the problem of analysis of semantic correctness of complicated artificial systems. The object-oriented approach gives perfect possibilities for construction of models of complicated systems, but only object-oriented languages with strict declarative semantics are suitable for logical analysis of the systems (not just for the programming).

Another very important area of research is programming of reactive computer systems working in conditions of nonmonotonic external environment. We suppose that logic object-oriented languages are very promising for this aim, because they have the model-theoretic semantics, that is invariant to unpredictable behavior of external environment.

The third problem is of maintenance of correctness of program systems. Currently the idea of "design by contract" based on logical description and analysis of pre- and post-conditions of operations on objects is popular in OOA. But we suppose that it is much more natural and easier to write the programs using object-oriented logic language.

Our approach to developing of the logic OOA is considered in the first section of the article. In second, third and fourth sections the following logical means developed within the framework of our project and implemented in Actor Prolog are considered: the classes and worlds, the mechanism of repetitive proof of subgoals (the "actor mechanism") and the underdetermined sets. The fifth section is about related works. In the conclusion we discuss the possibilities connected to the usage of developed logical means in the area of visual man-machine interfaces, decentralized artificial intelligence, object-oriented deductive databases and computer aided software engineering.

1 A Logical Interpretation of OOA

The relations between the logic programming and OOA have a long history and an extensive bibliography coverage (see, for example, [9, 34, 29, 25, 33]). Making analysis of this area of research we came to conclusion, that the idea of "object" of the imperative OOA doesn't have an univalent implementation in the logic. All attempts "to transfer" this idea to logic languages break it up to, at least, three separate aspects, that have the special definitional possibilities, theoretical problems and means of implementation:

1. *The structural aspect of OOA*: the "objects" in the imperative programming are the natural means of structuring the program. In logic programming this aspect refers to the logic program text structuring and the special facilities that control the search space.
2. *The dynamic aspect of OOA* reflects the possibilities related to the modification of the objects in the imperative OOA. This aspect causes the greatest difficulties in the theory of logic programming; there are very hard problems of integrating objects with logic programming – persistent vs. backtrackable state, for instance.
3. *The information aspect of OOA* is associated with the problem of description of complicated data structures.

We suppose that developing the *complete* object-oriented logic language requires a "logical interpretation of OOA" – a set of logical means covering (in the aggregate) all the definitional possibilities of all listed aspects of OOA. In Actor Prolog the following logical means are used for this purpose:

1. Classes, worlds and inheritance.
2. The mechanism of repetitive proof of subgoals.
3. Terms, including the so-called underdetermined sets.

The developed syntactic constructions are in fact the modified formulae of the first order predicate logic, that are only logical analogs of some imperative programming features: classes, actors, underdetermined sets, operation of destructive assignment and others. So, all the logical means of Actor Prolog have a strict and clear declarative semantics.

2 Classes, Worlds and Inheritance

In Actor Prolog the mechanism of classes is used to control the topology of the search space of the program.

By analogy to the classes in the imperative OOA the syntactic constructions (the "classes") for structuring the program text are used in Actor Prolog.

A class in Actor Prolog is a set of clauses. Just as in imperative programming languages each class has an unique name and all classes are elements of an inheritance hierarchy.

The concept "instance of class" of the imperative programming has an analog in Actor Prolog too. *The instances of classes* (the "worlds") in Actor Prolog are applications of classes, but imperative operation *new* (as, for example, in C++) doesn't exist in the language. In Actor Prolog instances of classes are built during the proof of special formulae, that are named "constructors".

The logical essence of the instances of classes in Actor Prolog implies an important difference from their imperative analog. Namely, the instances of classes in Actor Prolog are deleted during backtracking of the program. There are no analogs of imperative operation *delete* and concept "destructor" in Actor Prolog.

Another important feature of Actor Prolog is that in this language the "instance of class" and the "data item" are different concepts, contrary to such imperative object-oriented languages as Smalltalk and C++. The reason for this is that in an object-oriented language *the principle of uniqueness of instances of classes* is convenient. In other words, it is convenient to consider all instances of classes (even referring to the same class) as different entities, that, for example, cannot be unified. This principle does not allow to consider the compound terms of the language as instances of classes, because it would make senseless the resolution mechanism (so doing, we cannot unify two different instances of the same compound term $[1,2,3] \not\equiv [1,2,3]$, for instance).

So, the instances of classes and the data items are different concepts in Actor Prolog. And the instances of classes in the language are named "worlds" (instead of "objects") just for this reason.

The instances of classes are constituents of the search space during the execution of the program. The instance of class includes:

1. The clauses of the class and its ancestors in the inheritance hierarchy.
2. The set of "slots" of the instance.

The inheritance mechanism of Actor Prolog is analogous to similar mechanisms in the imperative object-oriented languages. But it has slightly different operational semantics: the hierarchy of inheritance determines the rules of search of the clauses of the program. For example, if the search space for some predicate p is an instance of class A, the search of the appropriate clause with heading p will be carried out among the clauses of class A, then among the clauses of the direct ancestor of class A, and so on.

Thus, *the overriding of clauses* doesn't exist in Actor Prolog, though it can be simulated using the non-logical built-in predicate cut '!'. Also *multiple inheritance* isn't used in Actor Prolog (there can be only one direct ancestor of the

class), because in the opposite case the search order for the clauses of parent classes would be not clear and should be determined *artificially*.

A slot is a "global variable" of an instance of class. The names of the slots are called "attributes of classes".

The attributes must be declared in all the classes, in which they are used. In the declaration of attributes the "initializers" defining the values of slots can be used. *The initializers* are terms, constructors of worlds or other attributes of the class.

Let's consider an example of class definition:

```
class 'ADDER' specializing 'ALPHA' is
a
b    = 0        -- The definition of the attributes of
c1   = 0        -- the class. The slots "b" and "c1"
sum             -- contain 0 by default.
c2
[               -- The clauses of the class.
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).
get_state(sum).
goal:-
     table(a,b,c1,sum,c2).
]
```

The class *'ADDER'* represents a complete binary adder. The attributes a, b and $c1$ designate the addends of the adder and the source carry bit. The attributes sum and $c2$ designate the sum and the target carry bit. The class *'ALPHA'* is the direct ancestor of the class *'ADDER'*.

The proof of the constructor (the construction of an instance of some class C) includes the following stages:

1. The *formation* of the instance of the class:
   (a) The construction of the appropriate search space.
   The search space will include the clauses of class C and also the clauses of all ancestors of class C.
   (b) The formation of the slots of the world.
   Each slot will get an initial value, if this slot has an appropriate initializer. In this case the initial value of the slot will be a data item or a world passed *only* the first stage of construction (the *formation*). If the slot doesn't have an initializer, it will get an anonymous variable as the initial value.
2. The proof of the predicate *goal* in all the worlds *formed* during the stage (1). The proof of the constructor will be considered as successful, if the proof of the predicate *goal* is successfully finished in *all* these worlds.

In our example the constructor

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

will create a new instance of the class *'ADDER'* and will set $Result = 0$.

The world that is a search space for a predicate can be indicated in any subgoal of a clause with the help of an attribute or constructor. Such subgoals are named "far calls of predicates". If the search space of a predicate of the clause is indicated by the constructor, the proof of this constructor takes place directly before the execution of the predicate, every time when the proof of the subgoal is carried out. Thus, during the execution of the program the search space can be created *dynamically.*

This idea is illustrated by the following example.

```
class 'PARALLEL_ADDER' is
input1      -- In this example the values of the slots
input2      -- of an instance of the class are
output      -- lists of integers.
[
goal:-
     loop(input1,input2,0,output).
loop([ ],[ ],_,[ ]):-!.
loop([A|R1],[B|R2],C1,[S|R3]):-
     ('ADDER', a= A, b= B, c1= C1, c2= C2) ? get_state(S),
     loop(R1,R2,C2,R3).
]
```

During the construction of an instance of the class *'PARALLEL_ADDER'*, a dynamic assembling of a parallel adder takes place. The components of the device are instances of the class *'ADDER'*, created during the proof of the predicate *loop*: the constructor of the instances of the class *'ADDER'* defines worlds, in which the far calls of the predicate *get_state* are carried out. The *dynamic* construction of the worlds in this example allows the creation of an adder of any necessary word length, depending on the length of input lists.

In the thesis [5] a theorem was proved that illustrates the logical interpretation of the structural and information aspects of OOA in Actor Prolog. The theorem states the logical correctness of the mechanism of classes considered above.

Theorem 1. *Any program written in Actor Prolog without actors and non-logical built-in predicates can be effectively transformed (there is a global syntactic transformation keeping the operational semantics) into the program in pure Prolog (or into the formula of Horn's subset of the first order predicate logic).*

Thus, the classes, the worlds, the underdetermined sets and other means reflecting the structural and information aspects of OOA in Actor Prolog are in fact only syntactic sugar and have a classical declarative semantics.

3 The Logical Actors and the Repetitive Proof

The idea of the repetitive proof of subgoals and the control strategy (the actor mechanism), that implements it, are the most interesting and most important elements of Actor Prolog [3, 6, 4, 5].

To describe these means of the language the following concepts will be necessary (see Fig. 1).

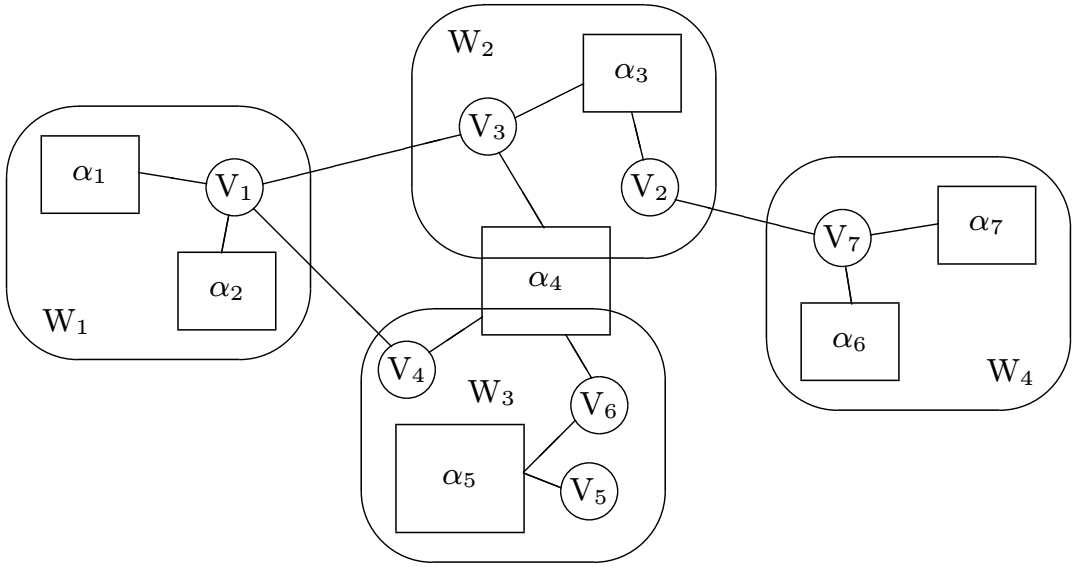

Fig. 1. The idea of repetitive proof of subgoals.

Within the framework of the ideology of Actor Prolog a logic program is considered as a theorem divided into the "logical actors". *The logical actors* are some subgoals of the theorem (α_1, ..., α_n on the figure), interacting through *common variables* (V_1, ..., V_m). The proof of this theorem is carried out in *an object-oriented search space* consisting of separate *worlds* (W_1, ..., W_k). For example, the box α_4 on the figure denotes an actor having subgoals, that are executed in the worlds W_2 and W_3.

The actor mechanism is an extension of the standard control strategy, based on the repetitive proof of actors, and implementing modifications of the values of the common variables. The purpose of the repetitive proof is the maintenance of soundness and completeness of proof of the theorem.

The interaction between the actors is implemented with the help of the "correct destructive assignment" operation. During this operation the necessary modifications of the values of common variables are carried out and then a repeated proof of actors depending on the old values of common variables is called. If the repeated proof of *all* these actors is finished with success, the execution of the operation also comes to the end with success, otherwise the usual backtracking of the logic program takes place.

Such coordination of logical actors with the help of the destructive assignment is carried out automatically at the moment of successful completion of a proof

of each actor. Also it can be called manually with the help of special built-in predicate ':='.

The built-in predicate ':=' implements the operation of the destructive assignment, modifying the values of common variables. The declarative semantics of this predicate is exactly the same as the semantics of the usual equality '=' in pure Prolog. But the operational semantics of the predicate ':=' is determined by the actor mechanism.

It is possible to illustrate this idea with the following example. Let's consider the behavior of the following fragment of Actor Prolog program (a special prefix @ is used in the language for definition of logical actors):

```
goal:–                          goal:–
    subgoal_a(X),                   @ subgoal_a(X),
    subgoal_b,                      subgoal_b,
    user_input(Y),                  user_input(Y),
    X := Y.                         X := Y.
subgoal_a(1).                   subgoal_a(1).
subgoal_a(3).                   subgoal_a(3).
subgoal_a(5).                   subgoal_a(5).
(a)                             (b)
```

Let's note, that in the real program an operation of reading from screen dialog or from a database could be used instead of the predicate *user_input*.

In the case (*a*) actors aren't used. In such cases Actor Prolog uses the standard control strategy as "usual" Prolog does.

So, during the proof of the goal statement the solution $subgoal\_a(1)$ will be found first. The variable X will get the value $X = 1$. Then subgoals $subgoal\_b$ and $user\_input$ will be executed. The user will enter the value of Y (for example, $Y = 5$). When the execution reaches the relation $X := Y$, there will be a failure because $1 \neq 5$. The backtracking will be called and the program will select a new fact $subgoal\_a(3)$. The following attempt (with $X = 3$) also will fail (let's suppose that user will enter $Y = 5$ again). And only the third attempt (when $X = 5$) will be successful. Let's note, that the subgoal $subgoal\_b$ was proved three times, though it had no relation to the variable X.

In the case (*b*) the subgoal @subgoal_a(X) will define a logical actor. Therefore the first attempt to execute the subgoal $X := 5$ will be finished with success. The result of this assignment will be a repeated proof of the subgoal $subgoal\_a(X)$; the operation of destructive assignment will "neutralize" the actor (will abolish results of its proof) and will prove it *again*. The fact $subgoal\_a(5)$ will be selected at this time.

We should note, that in the real program the repeated proof of an actor can "affect" other actors, that generally will call an avalanche process of elimination of inconsistencies between the logical actors.

Let's note, that the neutralization of actors has no relation to the idea of intelligent backtracking. The neutralization *is not some kind of backtracking*,

because the old outcomes of the proof of an actor are not rejected from *the stack* during its neutralization and repeated proof (as it takes place during the backtracking). So, the old outcomes can become again "acting", if the standard backtracking will happen in the program. This is the reason why some people call the idea of repetitive proof the "*anti*-backtracking".

We should note also, that there is an interdiction in the language for the neutralization of the actors, whose proof isn't finished yet. This rule averts the recursive neutralization of the actors, that can recycle the program.

The basic idea that drove the development of the actor mechanism was Hewitt's actor model of computations [14]. But the mechanism of repetitive proof of subgoals ensures the different principle of interaction between the actors.

Here are the most important differences:

1. An information interchange between the logical actors is carried out only through *the common variables.*
2. A logical actor does not know, what actors will be affected by the destructive assignment made by it. Thus, the interacting actors do not have to know each other "by name" and therefore the program in Actor Prolog is a *strongly distributed* system.
3. All the results of the neutralization and repeated proof are *backtrackable.* This is an extension of Hewitt's actor model too.
4. Another interesting property of the logical actors is *the absence of buffering of messages* between them. Generally, the common variables have a chance to get modified before the repeated proof of an actor depending on them will start.
5. The restriction of the logical actors is in that the results of the preceding proof of the logical actor are *inaccessible* after its neutralization and repeated proof are initiated. Our experience shows that it takes at least two logical actors to simulate an agent with the memory.

The detailed and complete description of the actor mechanism can be found in the paper [6] and in the thesis [5]. An abstract machine of Actor Prolog is defined in [5].

In the thesis [5] the following theorems about the soundness and completeness of the actor mechanism are proved:

Theorem 2. *(on the soundness of the actor mechanism) The mechanism of repetitive proof of Actor Prolog with the occur check and without the non-logical built-in predicates is a sound control strategy.*

Theorem 3. *(on the completeness of the search space) Any Actor Prolog program without the non-logical built-in predicates will find all the existing solutions, if an infinite computation does not arise during its execution.*

The definition of the declarative semantics of Actor Prolog program using the theorem 1:

Definition 1. *The declarative (model-theoretic) semantics of Actor Prolog program without the non-logical built-in predicates is the declarative semantics of the pure Prolog program corresponding (in the sense of the theorem 1) to the source program after deleting of all the prefixes @.*

So, all three aspects of OOA considered in the section 1 (structural, dynamic and information) are implemented in Actor Prolog with the help of pure logical means having a classical model-theoretic semantics.

4 The Underdetermined Sets

The information aspect of OOA is implemented in Actor Prolog with the help of several simple and compound terms: integers, reals, symbols, text strings, structures, lists and underdetermined sets. The last one is the most interesting type of terms of the language and should be considered more closely.

The underdetermined sets in Actor Prolog are syntactic constructions of a type

$$\{\text{a:X, b:17 | Rest}\} \text{ or } \{\text{b:Y, c:2, a:5}\} \ .$$

Generally, it is possible to consider such set as an "infinite" list of pairs "the_name_of_element: the_value_of_element", where the name of an element is a symbol or integer ≥ 0, and the value of an element is a variable or any other term. The construction $|Rest$ designates the "rest" of the set. If the variable $Rest = \{\}$, then no additional elements can be presented in the set – in this case it would be possible to write simply $\{a : X, b : 17\}$.

The algorithm of unification for underdetermined sets has been developed by the author. We will consider this algorithm by using the example of unification of two terms mentioned above. Let's note, that Actor Prolog uses the '==' as the equality operator as opposed to '=' in "usual" Prolog.

$$\{\text{a:X, b:17 | SetRest}\} == \{\text{b:Y, c:2, a:5}\}$$

During the unification the elements of the first underdetermined set will be compared with the elements of the second one in the correspondence to the given names. The result of the unification will become the substitutions $X = 5$, $Y = 17$, $SetRest = \{c : 2\}$.

The underdetermined sets are used in the language as terms and as atomic formulae. The correctness of this extension of Prolog can be easily shown: it is possible to transform underdetermined sets into constructions of the first order predicate logic [3, 6, 5].

Theorem 4. *(lemma of theorem 1) Any Actor Prolog program without the non-logical built-in predicates can be effectively transformed (there is a global syntactic transformation keeping the operational semantics) into Actor Prolog program without the underdetermined sets.*

We should note, that the underdetermined set cannot contain pairs with the identical names of elements. For example, the rest of the set $\{a : X, b : Y|Rest\}$ cannot be unified with any underdetermined set containing an element with the name a or b. So, the rest of the underdetermined set keeps a "negative" information about the constraints imposed on possible values of the underdetermined set during the execution of the program.

Thus, the underdetermined sets allow us to operate with unknown entities, specifying the boundaries of the domain to which they belong. So, an application of the underdetermined sets is *imitation of second order logic* in Prolog.

There is a special type of syntactic sugar in the language: the term F in the front of an underdetermined set – $F\{\ldots\}$ – denotes the additional element $'' : F$ of the set, where '' is a special symbol consisting of empty set of graphemes: $F\{\ldots\} \equiv \{'' : F, \ldots\}$.

Moreover, the syntax of the language allows the atomic formulae

F { sX:AX, sY:AY, ..., sN:AN | Rest } ,

that are equivalent to the "usual" atoms

'' ({ '':F, sX:AX, sY:AY, ..., sN:AN | Rest }) .

A call of such subgoal will perform an associative search in the database. And if an underdetermined set is given in *the heading* of a rule, such clause gets some properties of second order function.

Let's consider an example of usage of second order logic (see Fig. 2). This clause is used in Actor Prolog program for synthesis of algorithms – it states how to use the "if-fi" construction.

```
F { argument: A0, result: Z | Rest }:–
        A0 == { even: unknown | Pairs },
        A1 == { even: yes | Pairs },
        A2 == { even: no | Pairs },
        F { argument: A1, result: { dom: Y1 | _ } | Rest },
        F { argument: A2, result: { dom: Y2 | _ } | Rest },
        Z == { dom: 'if' ([ guard ( odd (A0), Y2),
                            guard ( even (A0), Y1) ]) }.
```

Fig. 2. An example imitating second order logic in Actor Prolog.

The rule should be used, if function F cannot be calculated with the argument $A0$, that can have both even and odd values. Then it is possible to represent the result of the function F as "if-fi" construction. For this purpose two new variables $A1$ and $A2$ are created. These variables have got all the properties (all the pairs) of the argument $A0$ except the parity indicator. $A1$ is declared as even and $A2$ is declared as odd. If the calculations of the function F with these arguments are completed with success, this implies that the construction "if-fi" is a valid result of the function F.

5 Related Works

The main novelty of Actor Prolog is the repetitive proof of actors, that allows implementation of the dynamical behavior of objects and destructive assignment.

There are a lot of modern logic languages implementing dynamical behavior of objects and destructive assignment (Visual Prolog [1], LogTalk [30], Prolog++ [18], SICStus objects [32] and others). However most of them are based on the non-logical means (like databases, blackboards and so on) and, thus, lose the declarative semantics (see, for example, the languages listed above). In contrast to these languages Actor Prolog supports a clear declarative semantics of the destructive assignment operation.

There are very interesting papers about using the transaction logic [11], linear logic [34, 23, 17], "temporal predicates" [19] and others to capture the notion of object modification. These approaches are pure logical and naturally fits into the logic programming setting (see, for example, languages Lolli [23] and Forum [17]). But Actor Prolog has the following differences:

1. Actor Prolog is based on the *classical* first order logic.
2. Actor Prolog isn't yet another means for the imitation of changes and destructive assignment, because the actor mechanism also reveals and eliminates *the contradictions* between the objects.

There are some approaches using daemons (Prolog++ [18], LogTalk [30]), blackboards (BinProlog [31]), "distributed objects" (Brain Aid Prolog [21]), database constraints [11] and so on to ensure the consistency between the objects. But these approaches aren't logical one. In contrast, Actor Prolog ensures the consistency, that is based on *the strict declarative semantics* of the objects, instead of using user defined rules or database constraints.

So, we claim that the "jump" to the new theory in case of the state change suits not only theoretical aspects, but also some practical demands: in fact, Actor Prolog *extends and generalizes* the object-oriented approach.

It is necessary to note also, that there are a lot of promising projects using Hewitt's actor formalism as a means for linking concurrent logic programming with OOP – I mean the so-called "process view" approach [9] to the logic OOP (Concurrent Prolog [20] and others). However Actor Prolog is irrelevant to the process view of objects.

The classes and instances of Actor Prolog are based on the well-known "clauses view" of logic OOP [9]. But Actor Prolog has the following features:

1. Slots (global variables) in Actor Prolog are based on *pure logical* means. For example, the slots can have unbound values (like the slots in the language OL(P) [28]). However the changes of slot state are implemented in Actor Prolog with the help of repetitive proof.
2. Actor Prolog implements *implicit* creation and elimination of instances of classes. So, Actor Prolog doesn't need imperative operation *new*, that looks quite unessential in the logic languages (see, for example, [1, 30, 18, 32]). It is important also, that the classes/instances mechanism of Actor Prolog

doesn't need any *garbage collection*, because the instances of classes are eliminated by the usual backtracking.
3. There are strong distinction between the instances of classes and data items in Actor Prolog. This feature allows *the uniqueness of instances of the classes* in the language.

The underdetermined sets are similar to *feature terms* of the language LIFE [22] and F-logic [2]. However the underdetermined sets have the following important difference – there is the notion "rest of set" in Actor Prolog. This simple feature is of great significance to the definitional power of the language:

1. One can determine whether underdetermined set can accept some *additional elements*. For example, the feature term $t(a \Rightarrow 7, b \Rightarrow 9)$ can be represented as the underdetermined set $t\{a : 7, b : 9|\_\}$ with the anonymous rest variable. But we cannot describe the set $\{a : 7, b : 9\}$ with the help of the feature terms.
2. The unification algorithm of Actor Prolog guarantees that *all names are unique* in the one underdetermined set. For instance, the underdetermined set $\{a : 10, b : 20|Rest\}$ separates the elements ($Rest$), that are not equal to a and b. Thus, one can assemble some new set using these elements, for example, $\{x : 100, y : 200|Rest\}$.
3. The underdetermined sets are *more simple notion* than the feature terms, because Actor Prolog doesn't use the type hierarchy and "tags".

The construction $\{\ldots|\ldots\}$ of Actor Prolog looks like *insertion* operator $\{\ldots|\ldots\}$ of so-called *list-like* representation of sets [10]. However there is an important difference: the underdetermined set $\{a : 100, b : 200|R\}$ states that the rest R has no common elements with the set $\{a : 100, b : 200\}$, while the insertion operator doesn't.

Let's note also, that the syntactic sugar of a type $F\{\ldots|Rest\}$ is a new useful means for *the higher-order logic programming* [26].

These properties of the underdetermined sets are very useful for the imitation of second order logic in Actor Prolog, though one can use any other kind of terms to perform an associative search in the program clauses. For example, there are very promising approaches based on the lambda terms (λProlog [16]) and "usual" sets (SPARCL [27], Flang [13] and others). "Partitions" of sets of the language SPARCL are similar to the construction $\{\ldots|\ldots\}$ of Actor Prolog. But in contrast to these approaches Actor Prolog doesn't use the lambda terms or constraint solving mechanism (as opposed to [27, 13, 10]). So, we suppose that the underdetermined sets are more simple, natural and practical means.

Conclusion

Let's look into some interesting properties of the presented approach and consider the possible practical applications of Actor Prolog.

First we must point out, that the control strategy of Actor Prolog (the mechanism of repetitive proof of subgoals) allows the correct implementation of the

destructive assignment in a logic language and thus is a new solution of *the frame problem* [24, 8], free of disadvantages of *the nonmonotonic logic systems* [12].

The repetitive proof is a quite simple idea. It seems, it is a new *basic principle* of logic programming (like recursion, unification, backtracking, parallelism, lazy evaluations, etc.). Thus, one can combine repetitive proof with other basic principles to make some new control strategies. For example:

- The repetitive proof is combined with *SLD resolution* and *backtracking* in Actor Prolog. So, all the results of repeated proof are backtrackable.
- One can combine repetitive proof with *parallelism* – the concurrent execution of some logical actors is a quite essential idea.
- One can combine repetitive proof with *a constraint solving mechanism.*
- One can combine repetitive proof with *lazy evaluations.*
- One can combine repetitive proof with *persistence* (it's more easy to implement this idea, than the persistent backtrackable states).

So, the precise definition of actor mechanism (the abstract machine) [5] is only one of possible implementations of the repetitive proof in logic programming.

The mechanism of repetitive proof can work in *strongly distributed systems* in the conditions of inconsistency and delayed updating of information. Thus, the developed control strategy is a powerful tool of *truth maintenance* and can be used in logic programming of *the open systems* [15, 8].

There is no doubt that the theoretical properties of the mechanism of repetitive proof mentioned above open some new interesting possibilities in the area of *deductive databases*, *decentralized artificial intelligence* and *WWW/Internet logic programming.*

Within the framework of the project the problem of making of *intellectual visual man-machine interface* was considered as an application.

From the logical point of view, it is very convenient to consider interaction between a person and a computer as a proof of some theorem, in which the person is modifying the conditions of the task and the machine ensures the correctness of the proof. In the ideology of Actor Prolog user's input is interpreted as a correct destructive assignment calling repeated proof of actors of the logic program. This idea was implemented as a set of predefined classes of modeless input/output in Actor Prolog, within the framework of RFBR project 95-01-00822. I think that the next step in this direction will be the programming of *virtual reality* objects with the help of mechanism of repetitive proof and Actor Prolog.

Another very interesting area of application of Actor Prolog is the logical object-oriented *description and analysis of complicated information systems.* In the thesis [5] *the method of interactive functional modeling of information systems* was developed on the basis of Actor Prolog. The central idea of this method is the use of the object-oriented logic language for description and analysis of SADT diagrams [7].

Currently we launch several new projects and welcome a co-operation concerning the logical *image processing*, *pattern recognition* and designing of *the decisions support systems.*

Acknowledgements

I would like to thank Yury V. Obukhov, Alexander V. Boudarov, Alexander F. Polupanov, Vladimir A. Zakharov, Michael V. Zakharyashev, Irina V. Gorskaya and all my friends and colleagues, who provided me with their help and support.

References

1. *Visual Prolog Version 5.0. Language Tutorial.* Prolog Development Center, Copenhagen, Denmark, 1997. (http://www.pdc.dk).
2. A. A. Fernandes, N. W. Paton, M. H. Williams, A. Bowles. Approaches to Deductive Object-Oriented Databases. *Information and Software Technology*, 34(12):787–803, 1992.
3. A. A. Morozov. Actor Prolog. *Programmirovanie*, (5):66–78, 1994. In Russian.
4. A. A. Morozov. Actor Prolog. In *The Discrete Models in the Theoretics of Control Systems: Proc. of the II Int. Conf.*, Moscow, Russia, 1997. Dialog-MSU. In Russian. (http://www.cplire.ru/Lab144/report1.html).
5. A. A. Morozov. *The Logical Analysis of Functional Diagrams during the Interactive Designing of Information Systems.* PhD dissertation, IRE RAS, Moscow, Russia, June 1998. 199 pp. In Russian. (http://www.cplire.ru/Lab144/auto.html).
6. A. A. Morozov, Yu. V. Obukhov. Actor Prolog. Programming Language Definition. Preprint 2(613) of 14.06.96, IRE RAS, Moscow, Russia, June 1996. 57 pp. In Russian. (http://www.cplire.ru/Lab144/index.html).
7. A. A. Morozov, Yu. V. Obukhov. Interactive Semantic Analysis of SADT Diagrams of the Information Systems by the Means of Object Oriented Logic Programming. In *Development and Application of Open Systems: Proc. of the IV Int. Conf.*, pages 61–64, Nizhny Novgorod, Russia, 1997. In Russian. (http://www.rapros97.nnov.ru/reports/9.html).
8. A. A. Morozov, Yu. V. Obukhov, A. Ja. Olejnikov. Logic Programming of Open Systems. In *Proc. of the XI Int. Conf. on Logic, Methodology and Philosophy of Science*, volume 2, pages 153–156, Moscow-Obninsk, Russia, 1996. In Russian. (http://www.cplire.ru/Lab144/obninsk.html).
9. A. Davison. A Survey of Logic Programming-based Object Oriented Languages. Technical Report 92/3, Dep. of Computer Science, University of Melbourne, Melbourne, Australia, January 1992. (http://www.cs.mu.oz.au/tr_db/mu_92_03.ps.gz).
10. A. Dovier, C. Piazza, E. Pontelli, G. Rossi. On the Representation and Management of Finite Sets in CLP-languages. Technical report NMSU-CSTR-9715, New Mexico State University, USA, December 1997. (ftp://ftp.dimi.uniud.it/pub/dovier/jicslp98.dvi.gz).
11. A. J. Bonner, M. Kifer, M. Consens. Database Programming in Transaction Logic. In *Database Programming Languages*, pages 309–337. Springer-Verlag, 1994.
12. A. Thayse, P. Gribomont, G. Louis, D. Snyers, P. Wodon, P. Gochet, E. Grégoire, E. Sanchez, P. Delsarte. *De la Logique Classique à la Programmation Logique*, volume 1 of *Approche Logique de l'Intelligence Artificielle*, chapter 4. Dunod Informatique, Paris, 1988.
13. A. V. Mantsivoda. *Σ-Programming and Problems of Discrete Optimization.* Irkutsk State University, Irkutsk, Russia, 1994. In Russian.
14. C. Hewitt. Viewing Control Structures as Patterns of Passing Messages. *Artificial Intelligence*, 8(3):323–364, 1977.

15. C. Hewitt. Open Information Systems Semantics for Distributed Artificial Intelligence. *Artificial Intelligence*, 47(1/3):79–106, 1991.
16. D. Miller. Lambda Prolog. http://www.cse.psu.edu/~dale/lProlog/index.html, May 1997.
17. D. Miller. The Forum Specification Language. http://www.cis.upenn.edu/~dale/forum/, May 1997.
18. D. Pountain. Adding Objects to Prolog. *Byte*, 15(8), 1990.
19. D. W. Embley, S. W. Liddle, Y.-K. Ng. On Harmonically Combining Active, Object-Oriented, and Deductive Databases. In *ADBIS'96: Proc. of the Third Intern. Workshop on Advances in Databases and Information Systems*, pages 21–30, Moscow, Russia, 1996. MEPhI.
20. E. Shapiro, A. Takeuchi. Object Oriented Programming in Concurrent Prolog. *New Generation Computing*, (1):25–48, 1983.
21. F. Bergmann, M. Ostermann, G. von Walter. Brain Aid Prolog. http://flp.cs.tu-berlin.de/~fraber/bap.html, 1999.
22. H. Aït-Kaci, B. Dumant, R. Meyer, A. Podelski, P. Van Roy. *The Wild LIFE Handbook.* Paris Research Laboratory, prepublication edition, March 1994. (ftp://ftp.info.ucl.ac.be/pub/life/Life_manual.ps.Z).
23. J. Hodas. Lolli: A Linear Logic Programming Language. http://www.cs.hmc.edu/~hodas/research/lolli, July 1995.
24. J. McCarthy. The Generality in Artificial Intelligence Systems. In Ashenhurst R., editor, *ACM Turing Award Lectures: The First Twenty Years (1966 to 1985)*, ACM Press Anthology Series. ACM Press, New York, 1987.
25. J. P. Bowen. Logic Programming. Oxford University Archive. http://www.comlab.ox.ac.uk/archive/logic-prog.html, August 1999.
26. L. Naish. Higher Order Logic Programming in Prolog. Technical report 96/2, Department of Computer Science, University of Melbourne, Melbourne, Australia, February 1996. (http://www.cs.mu.oz.au/~lee/papers/ho/).
27. L. Spratt, A. Ambler. A Visual Logic Programming Language Based on Sets and Partitioning Constraints. In *Proc. of the 1993 IEEE Symposium on Visual Languages*, June 1993. (http://www.designlab.ukans.edu/~spratt/papers/SPARCL_93.dir/SPARCL_93.html).
28. M. P. J. Fromherz. *OL(P): Object Layer for Prolog – Reference Manual.* Xerox Corporation, 1993. (ftp://parcftp.xerox.com/pub/ol/ol.tar.Z).
29. M. Schneider, R. Katz. Object-Oriented Language: Prolog. http://www.cetus-links.org/oo_prolog.html, July 1999.
30. P. Moura. *Logtalk Object-oriented Programming in Prolog.* Centre for Informatics and Systems, University of Coimbra, Coimbra, Portugal, July 1999. (http://www.ci.uc.pt/logtalk/logtalk.html).
31. P. Tarau, V. Dahl. Mobile Threads through First Order Continuations. In *Proc. of APPAI-GULP-PRODE'98*, Coruna, Spain, July 1998. (http://www.cs.unt.edu/~tarau/research/98/GULP98.html).
32. Swedish Institute of Computer Science, Kista, Sweden. *SICStus Prolog User's Manual. Release 3.7.1*, October 1998. (http://www.sics.se/isl/sicstus.html).
33. V. A. Petukhin. Slava Petukhin's Logic Programming Page. http://www.isu.ru/~slava, January 1999.
34. V. Alexiev. Mutable Object State for Object-Oriented Logic Programming: A Survey. Technical report TR93-15, Dep. of Computing Science, University of Alberta, Alberta, Canada, 1993. (ftp://ftp.cs.ualberta.ca/pub/oolog/state.ps.Z).