Towards the Distributed Logic Programming of Intelligent Visual Surveillance Applications

Alexei A. Morozov[1,2(✉)], Olga S. Sushkova[1], and Alexander F. Polupanov[1]

[1] Kotel'nikov Institute of Radio Engineering and Electronics of RAS, Mokhovaya 11-7, Moscow 125009, Russia
morozov@cplire.ru, {o.sushkova,sashap55}@mail.ru
[2] Moscow State University of Psychology and Education, Sretenka 29, Moscow 107045, Russia
http://www.fullvision.ru/actor

Abstract. An extension of the Actor Prolog language with the ability of distributed logic programming is demonstrated. This language extension is developed for experimenting with distributed logic programming and declarative agent approach to intelligent visual surveillance. An approach to resolving the contradiction between the strong typing of Actor Prolog and the independence of software agents is proposed. Remote calls of Actor Prolog predicates are implemented using the object-oriented features of Actor Prolog, translation of Actor Prolog to Java, and the Java RMI protocol. An example of logic program communication based on the remote predicate calls is examined.

1 Introduction

A distributed version of the Actor Prolog language [1,2] described in this paper is intended for experimenting with the declarative agent approach to the intelligent visual surveillance. The concept of the multi-agent programming came to the field of intelligent visual surveillance from Artificial Intelligence [3–8]. The idea of the multi-agent approach to the visual surveillance is in that the intelligent visual surveillance system consists of communicating programs (agents) that have the following properties: autonomy (they operate without direct control from users and other agents), social ability (they can co-operate to solve the problem), reactivity (they perceive the environment and respond to external events), and pro-activity (they demonstrate a goal-directed behavior). Theoretically speaking, the multi-agent approach can provide flexibility, reliability, and openness of the intelligent visual surveillance systems [9,10]. For instance, let us imagine that stages of video analysis are implemented by a set of agents. Then, the visual surveillance system can be easily extended by an additional method of abnormal behavior recognition without modification of its agents and even without suspending its work. One just needs to insert in the system a new agent that can utilize results of other agents and transfer his own results to others.

O. Pichardo-Lagunas and S. Miranda-Jiménez (Eds.): MICAI 2016, Part II, LNAI 10062, pp. 42–53, 2017.
DOI: 10.1007/978-3-319-62428-0_4

In recent decade, the declarative approach to the development of the multi-agent systems is recognized as a basic idea in this research area; a set of excellent declarative multi-agent platforms and languages are developed and implemented [5,7]. Unfortunately, in the framework of the intelligent visual surveillance systems, the agents are to perform very specific operations on big arrays of binary data that are out of the framework of the conventional symbolic processing operations typical for declarative languages. Thus, there is a reason for development of the new means of the multi-agent-programming for experimenting with intelligent visual surveillance systems. The distinctive feature of our approach is in that we implement the intelligent video analysis using the concurrent object-oriented language Actor Prolog and a compiler of Actor Prolog into pure Java [11].

Previously, the object-oriented logic approach to the intelligent visual surveillance was reported in [12–16]. It was demonstrated that the translation of the object-oriented logic language into Java yields a sufficiently fast executable code for real time video analysis and detection of complex patterns of the abnormal people behavior. This approach can be extended to the distributed visual surveillance, because the Actor Prolog language is indeed an object-oriented language and can be easily adapted to the distributed programming framework even without modifications of the syntax. The only problem to be solved is the incorporation into the language the ability of remote procedure calls.

The Actor Prolog language differs from other state-of-the-art Prolog-based agent languages like Jason [17] and 2APL [18] in that it is not based on the BDI model and it does not directly offer high-level features such as planners and agent communication languages that might be expected for a multi-agent language. Actor Prolog is rather a more high-performance object-oriented logic language that is a base for implementation of real time multi-agent application platforms.

The paper is organized as follows. The problem of the contradiction between the strong typing of the Actor Prolog language and the independence of the software agents is discussed in Sect. 2. The type system of the distributed Actor Prolog is considered in Sect. 3. An example of the remote communication between two independent Actor Prolog programs is examined in Sect. 4.

2 The Problem of the Strong Typing in Multi-agent Systems

A term “remote procedure call” is usually associated with the OMG GORBA, Java RMI, or MS DCOM protocols. This meaning of the term is relevant to the topic, because the remote predicate calls are implemented in the distributed Actor Prolog using the Java RMI protocol. At the same time, the term is linked with the general problems of the logic language design and implementation in the context of the agent logic programming.

It is known that interactions between independent agents are very hard to handle for strongly typed object systems [19]. The main problem to be resolved

in the course of adapting Actor Prolog to multi-agent paradigm is the contradiction between the strong type system of the language and the idea of independency of the software agents. The strong type system is an important feature of the language and is necessary for generation of fast and reliable executable code [11,13]. The problem is in that one needs to transfer information about the data types between the software units to implement their link and static type-checking. This kind of information exchange between the software agents is definitely undesirable, because it decreases the autonomy of the agents and complicates the agent life cycle. In this paper, another solution of the problem is proposed; the type system of the Actor Prolog language is partially softened to allow a dynamic type-checking (instead of the static one) in some restricted cases linked with the inter-agent communications.

Another problem that is close to the topic, but is still different, is a combination of the object-oriented paradigm and the strong typing. It was recognized earlier, that types are useful for formalizing and maintaining object interfaces, though types are orthogonal to objects and their integration is not a simple deed [20]. The Actor Prolog language supports both types (domains) and classes/objects. A distinctive feature of the language is in that the "object" and "data item" notions were clearly separated in the language [1]. The language has the strong type system that supports various kinds of simple and composite data items like numbers, structures, lists, etc. At the same time, Actor Prolog supports also classes based on the "clauses view" of logic OOP [21]. The instances of classes (so-called "worlds") can be processed like standard Prolog terms; they can be passed as arguments to predicates and can be included in composite terms. However, special rules are used for unification of variables containing instances of classes and special means are to be developed for interchange of these kinds of terms between the distributed agents.

In the Actor Prolog language, different instances of classes are treated always as different entities, that is, unification of two worlds succeeds if and only if these worlds are the same instance of a class. The interface of the class contains all necessary information about its methods including names, arity, flow patterns, and types of arguments. The information about the determinancy/non-determinancy of the methods is also included in the interface. There are three keywords in the languages for the declaration of the determinancy of the methods: *determ*, *nondeterm*, and *imperative*. The *nondeterm* keyword informs compiler that there are no restrictions on the behavior of methods and they can produce several answers in the case of backtracking and/or terminate with failure. The *determ* keyword means that methods can produce just one answer or terminate with failure. The *imperative* keyword imposes the hardest restrictions on the methods: the predicates must succeed and produce one answer; this means that the predicate operates indeed as a usual procedure in an imperative language. All these restrictions are checked by the compiler during the translation of the program.

The description and usage of the class interfaces are complicated a bit in Actor Prolog by the fact that the language supports concurrent processes and

two different kinds of method invocations: plain and asynchronous. The processes are a special kind of class instances; they are defined using double round brackets in the class instance constructors [2]. The plain method invocation is a usual predicate call of standard Prolog; the predicate can be invoked in a given world using the "?" prefix. The asynchronous method invocations are indicated by special prefixes "<-" and "<<". Only this kind of predicate calls is applicable for the processes; an attempt to implement a plain predicate call in a concurrent process will always terminate with a failure. The *internal* keyword is introduced in the language to facilitate optimization of the logic programs. This keyword informs the compiler that a given attribute of a class always contains a plain class instance, but not a process, that is important for analysis of predicate determinancy. Obviously, we will focus on the asynchronous method invocations in this paper, because class instances obtained from another logic program are processes that operate concurrently in relation to the invoking logic program.

Ordinarily, a standard static type-checking is implemented in the distributed version of Actor Prolog. The dynamic type-checking is implemented only if a method is to be called in an object (an instance of Actor Prolog class) that is originated from another logic program and is transferred somehow to the logic program under consideration. The verification of the remote predicate call includes the following operations:

1. One checks the name and the arity of the predicate. The predicate with the target name and arity is to be found in the object.
2. One checks the flow pattern of the predicate. The Actor Prolog language supports explicit declaration whether the argument is input or output; the flow directions of all the arguments in the predicate call are checked.
3. One checks so-called structural match of domains of all the arguments (this is a kind of dynamic type-checking).

The structural match of the domains means that graphs representing the data structures belonging to domains have to be equivalent, but not the names of the domains.

3 A Strong Type System in the Distributed Actor Prolog

Let us consider briefly the type system of the Actor Prolog language and the structural matching rules associated with various kinds of simple and compound data types (domains).

Actor Prolog supports the following simple data types: integer, real, symbol, and string. The difference between the integers and real numbers is in that the real numbers contain a dot. The difference between the symbols and strings is in that the symbols are represented by integer codes internally, but not by the text, during the execution of the program. On the syntax level, the symbols are enclosed in apostrophes and the strings are enclosed in quotes. Here is an example of using these built-in data types for definition of user data types:

```
DOMAINS:
Year            = INTEGER.
Height          = REAL.
Color           = SYMBOL.
Message         = STRING.
```

During the structural matching, the integers can match only the integers, the reals can match only the reals, etc. No automatic type conversions are allowed.

Actor Prolog supports so-called numerical ranges and enumerations. A range type can be defined using the integer or real bounds, for example:

```
Hour            = [0 .. 24].
Angle           = [0.0 .. 360.0].
```

The procedure of structural matching checks the exact equality of the bounds of the numerical range types to be compared. The only exception is that the real range bounds can slightly differ in accordance with the real number precision given in the translator options.

An enumeration type can be defined using a set of constants of any simple types, for instance:

```
Hour            = 0; 1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12.
Color           = 'Red';'Blue';'Green'.
```

The structural match of two enumerations means that these types include the same sets of elements.

Note that a type definition can include a set of names of other types in the language; this is a basic difference of the Actor Prolog type system from analogous type systems in the Turbo/PDC Prolog family [22]. For instance, an argument of the following data type can transmit both integer and real values:

```
Numerical       = INTEGER; REAL.
```

Generally speaking, a type definition can refer to other data types. The structure matching procedure considers all the type definitions and compares the corresponding sets of elements that can include simple domains, literals, and composite types.

There are three kinds of composite types in Actor Prolog, namely: structures, lists, and so-called underdetermined sets [1]. The structure type definition consists of a functor and arguments enclosed in round brackets, for example:

```
AppointedDate = date(Year,Month,Day).
```

The structural matching procedure checks whether two structure domain definitions contain the same functor and the same number of arguments. Then, the structural matching of the types of all corresponding arguments is implemented.

The lists are a separate type in Actor Prolog, but not a kind of the structures. The list type definition contains a name of element type and an asterisk, for instance:

```
Dates          = AppointedDate*.
```

The structural matching procedure checks the types of the elements of the list types.

The definition of an underdetermined set type in Actor Prolog contains an unordered set of named pairs enclosed in braces. Every pair contains the identifier of the pair and the type of the argument, for example:

```
Customer       = {name: STRING, birthday: Date, age: INTEGER}.
```

The structural matching procedure compares the types of all corresponding pairs in the definitions of undetermined set types. The types must contain the pairs of the same names, but the order of the pairs is insignificant.

There are two exotic data types in Actor Prolog: so-called anonymous type "_" and so-called "any set" type {_}. The former type indicates that a predicate accepts terms of any types; it is useful for the definition of read/write procedures, etc. The second data type is used for the definition of predicates/attributes that accept terms of any types, but only in a form of an underdetermined set, for instance:

```
HTTP_ContentParameters = {_}.
```

By the rules of the structural matching, the anonymous type matches only the anonymous type and the "any set" type matches only the "any set" type.

All the rules described above are applicable to both the static and dynamic type-checking. A single difference relates to the structural matching data types that contain class names. The point is that a type definition in Actor Prolog can include the name of a class enclosed in round brackets, for example:

```
MessageHandler = ('MyClass').
```

This type definition means that a term of the *MessageHandler* type can be an instance of the *MyClass* class. This instance can be a plain world or a concurrent process of the class. The definition tells nothing about the concurrent execution of the class instance, but does not prohibit this kind of class usage too. Actor Prolog considers this world data type as a simple one. The compiler of non-distributed Actor Prolog guarantees that a term of this type is an instance of the *MyClass* class or an instance of a class that inherits the *MyClass* class; this rule is softened in the distributed Actor Prolog.

The distributed Actor Prolog checks whether an instance of a class belongs to the class pointed in the type definition only if this class is defined in the same logic program (i.e., it is technically possible to check it). An instance of any external class obtained from another logical agent (program) can be freely assigned to a variable/predicate argument of any type that includes a class name. Thus, the structured matching algorithm allows matching of any world types; the names of classes in the type definitions are simply ignored if the classes are defined in different logic programs (agents).

In distributed Actor Prolog, an instance of a class can be transferred to another logic program somehow and be accepted without the check of the interface if the accepting program expects to accept an instance of some class. A real check of the class interface is to be performed when the accepting program try to invoke a method in the external object. In this case, the structural matching procedure described above is to be performed, that can confirm suitability of the object or yield a runtime error. Obviously, the implementation of this check requires information on the origin of all the objects in the logic program. Thus, distributed Actor Prolog keeps an internal table of all the class instances created during the program execution and transferred outside. Another internal table contains all the objects accepted somehow from other logic programs. These tables allow Actor Prolog to distinguish clearly the instances of own and external classes and use this information in the structured matching algorithm.

Thus, the multi-agent interaction in Actor Prolog is based on the fusion of dynamic and static typing. The static type-checking and standard features of a nominative type system are implemented for all the own worlds like in the conventional Actor Prolog. At the same time, the dynamic type-checking and elements of a structural type system are implemented for all the external worlds. We consider the type system of Actor Prolog as a combined type system. This type system ensures the advantages of the static type-checking for the high-performance code generation and the flexibility of the dynamic type-checking that is necessary for the multi-agent systems programming.

4 An Example of the Logical Agent Communication

Let us consider an example of the remote predicate call. Suppose there are two agents: *Recognizer* and *Observer* (see Fig. 1). These agents should co-operate to search and recognize people in a video scene. Suppose that *Recognizer* controls its own pan-tilt-zoom (PTZ) camera and can identify a person in given co-ordinates. *Observer* can analyze behavior of people and calculate co-ordinates of the persons to be identified. Suppose that these logic programs are different agents that should establish a link dynamically and exchange information to solve the problem.

First, let us define a schema of the *Recognizer* logic program. A logic program that is defined below creates an instance of a class and saves it in a file to be accessible for other programs. Other program can read this class instance and implement a remote call of a predicate defined in the *Recognizer* program. Let the external program transmits co-ordinates of a person to be identified and the *Recognizer* logic programs accept this information and simply print it in the screen for the sake of simplicity.

In accordance with the semantics of the Actor Prolog language, the execution of the program begins with creation of an instance of the *Main* class. In the program under consideration, the *Main* class is an instance of the built-in *Console* class that implements a text window control:

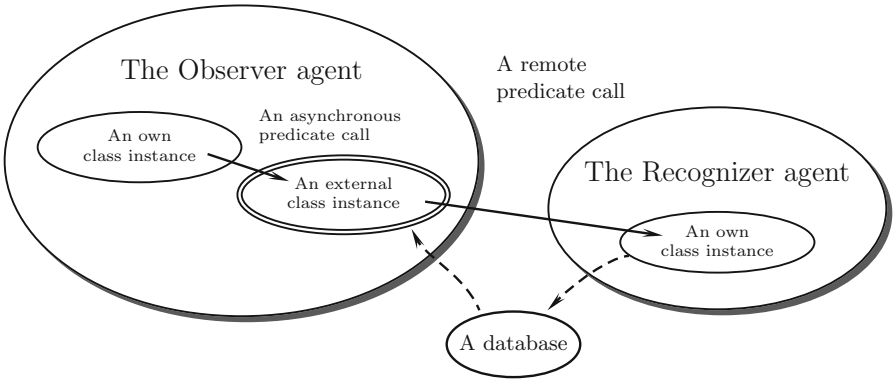

Fig. 1. An example of two co-operating agents. The *Recognizer* agent publishes an instance of a class in the external database. Then, the *Observer* agent obtains this class instance and sends an asynchronous message to this class instance using Java RMI.

```
class 'Main' (specialized 'Console'):
external_file     = ('DataExchange');
[
PREDICATES:
intruder_coordinates(REAL,REAL)     - (i,i);
MODEL:
?intruder_coordinates(X,Y).
```

The *Main* class contains a single slot named *external_file*. The value of this slot is an instance of the *DataExchange* class. The *DataExchange* class implements the data exchange using a built-in database for simplicity, since this is the simplest way of external file control in the Actor Prolog language.

There is a single predicate definition in the *PREDICATES* section. The *intruder_coordinates* predicate has two input real arguments. This predicate is never called directly inside the *Recognizer* logic program, that is why one should indicate in the *MODEL* section that this predicate is to be invoked with two arguments. Otherwise, the translator will discard this predicate during the optimization of the code.

The *CLAUSES* section of the *Main* class contains the definitions of the *goal* and *intruder_coordinates* predicates:

```
CLAUSES:
goal:-!,
      external_file ? insert(self),
      external_file ? save("g:/SharedData.db"),
      writeln("I wait for intruder co-ordinates...").
intruder_coordinates(X,Y):-
       writeln("X=",X,"Y=",Y).
]
```

The *goal* predicate is called automatically during the creation of the *Main* class instance. This predicate inserts the instance of the *Main* class into the *DataExchange* database using the *insert* built-in method and the *self* keyword. Then it records the database content to the file using the *save* built-in method and writes the message in the screen: "I wait for intruder co-ordinates..." The *intruder_coordinates* predicate is to be invoked from outside using the remote call protocol. This predicate simply writes the co-ordinates in the screen.

There is yet another class definition in the text of the *Recognizer* program. The *DataExchange* class inherits methods from the *Database* built-in class that implements a simple database management system. There is a definition of the *Target* domain in the *DOMAINS* section of the *DataExchange* class. This definition is necessary in order to inform the database management system about the type of data to be stored in the *DataExchange* class. It is declared that the *Target* type includes instances of the *Main* class.

```
class 'DataExchange' (specialized 'Database'):
[
DOMAINS:
Target      = ('Main').
]
```

Let us consider the *Observer* logic program. Suppose this program should obtain an instance of an external class from the file and send to this object a message containing co-ordinates of a person to be identified. The *Main* class of this program inherits methods from the *Console* built-in class too.

```
class 'Main' (specialized 'Console'):
file        = ('InternalDatabase');
[
PREDICATES:
send_coordinates('AcceptingAgent')  - (i);
CLAUSES:
goal:-
      file ? load("g:/SharedData.db"),
      file ? find(ExternalObject),!,
      send_coordinates(ExternalObject).
send_coordinates(ExternalObject):-
      ExternalObject << intruder_coordinates(125.009,1107.144),
      writeln("The information is sent...").
]
```

The *Main* class includes the *file* slot that contains an instance of the *InternalDatabase* class. There is a definition of the *send_coordinates* auxiliary predicate in the *PREDICATES* section. This predicate has one input argument that should contain an instance of a class that inherits methods from the *AcceptingAgent* interface defined below. The *goal* predicate acquires information from the external file using the *load* built-in method of the *InternalDatabase* database. Then it takes the *ExternalObject* world from the

database and transmits this class instance to the *send_coordinates* predicate. The *send_coordinates* predicate implements an asynchronous predicate call in the *ExternalObject* world and writes the text message in the screen: "The information is sent..." Note that the asynchronous call will be implemented using the remote call protocol, because the variable *ExternalObject* contains an object that originates from another logic program. The dynamic type-checking will be implemented during the call.

The *AcceptingAgent* interface describes methods that are expected to be supported by the collaborator of the *Observer* agent. Note that this interface links up in no way with the classes/interfaces of the *Recognizer* agent:

```
interface 'AcceptingAgent':
[
PREDICATES:
intruder_coordinates(REAL,REAL)     - (i,i);
]
```

The *InternalDatabase* auxiliary class is defined in the similar way as the *DataExchange* class in the *Recognizer* agent. The single difference is in that the *Target* domain includes instances of classes that inherit the *AcceptingAgent* interface.

```
class 'InternalDatabase' (specialized 'Database'):
[
DOMAINS:
Target      = ('AcceptingAgent').
]
```

Let us execute the *Recognizer* logic program. The program will create the *SharedData.db* file in the *g*: disk and write the text in the screen:

```
I wait for intruder co-ordinates...
```

The *SharedData.db* file is text one, because the *Database* class stores the information in a user-readable format. The file contains something like this (the text is reduced):

```
('feffiimuyf2uhb8pqbhesx ... rlp1l8o2mq9vmyzk2o7t4');
```

The alphanumeric code enclosed in the apostrophes and round brackets is a usual Actor Prolog term, namely, a text representation of a class instance. On the technical level, this is an encoded instance of a Java RMI stub that refers to the class instance. Note that this format of data exchange is appropriate for any type of data transfer protocols including E-Mails.

Let us launch the *Observer* agent now. This logic program will read the class instance from the *SharedData.db* external file, implement a remote predicate call in the *Recognizer* agent, and write the text message:

```
The information is sent...
```

The *Recognizer* agent will accept the remote predicate call and write the acquired co-ordinates in the screen:

```
I wait for intruder co-ordinates...
X= 125.009 Y=1107.144
```

This example illustrates the basic schema of agent data exchange in distributed Actor Prolog using the remote predicate calls, dynamic type-checking, and some technical details of the data encoding.

5 Conclusions

The extension of the Actor Prolog language with the ability of distributed logic programming was demonstrated. The idea of a combined type system which provides a new solution of the problem of the strong typing in the multi-agent systems was proposed and examined. This type system ensures the advantages of the static type-checking for the generation of the fast executable code and the flexibility of the dynamic type-checking that is necessary for the multi-agent systems design. It was implemented in the distributed version of the Actor Prolog language that gives new means for experimenting with the multi-agent logic programming. We suppose that these means open new prospects for the development of real time logical multi-agent systems and practical applications of the logic programming in the intelligent visual surveillance.

This research is supported by the Russian Foundation for Basic Research, grant No. 16-29-09626 (please see our Web Site [23] for details).

References

1. Morozov, A.A.: Actor Prolog: an object-oriented language with the classical declarative semantics. In: Sagonas, K., Tarau, P. (eds.) IDL 1999, France, Paris, pp. 39–53 (1999)
2. Morozov, A.A.: Logic object-oriented model of asynchronous concurrent computations. Pattern Recogn. Image Anal. **13**, 640–649 (2003)
3. Russell, S., Norvig, P.: Artificial Intelligence. A Modern Approach. Prentice-Hall, London (1995)
4. Shen, W., Hao, Q., Yoon, H., Norrie, D.: Applications of agent-based systems in intelligent manufacturing: an updated review. Adv. Eng. Inform. **20**, 415–431 (2006)
5. Baldoni, M., Baroglio, C., Mascardi, V., Omicini, A., Torroni, P.: Agents, multi-agent systems and declarative programming: what, when, where, why, who, how? In: Dovier, A., Pontelli, E. (eds.) A 25-Year Perspective on Logic Programming. LNCS, vol. 6125, pp. 204–230. Springer, Heidelberg (2010). doi:10.1007/978-3-642-14309-0_10
6. Gascueña, J., Fernández-Caballero, A.: On the use of agent technology in intelligent, multisensory and distributed surveillance. Knowl. Eng. Rev. **26**(2), 191–208 (2011)

7. Bădică, C., Braubach, L., Paschke, A.: Rule-based distributed and agent systems. In: Bassiliades, N., Governatori, G., Paschke, A. (eds.) RuleML 2011. LNCS, vol. 6826, pp. 3–28. Springer, Heidelberg (2011). doi:10.1007/978-3-642-22546-8_3
8. Kravari, K., Bassiliades, N.: A survey of agent platforms. J. Artif. Soc. Soc. Simul. **18**, 191–208 (2015). http://jasss.soc.surrey.ac.uk/18/1/11.html
9. Vallejo, D., Albusac, J., Castro-Schez, J., Glez-Morcillo, C., Jiménez, L.: A multi-agent architecture for supporting distributed normality-based intelligent surveillance. Eng. Appl. Artif. Intell. **24**, 325–340 (2011)
10. Ejaz, N., Manzoor, U., Nefti, S., Baik, S.: A collaborative multi-agent framework for abnormal activity detection in crowded areas. Int. J. Innov. Comput. Inf. Control **8**, 4219–4234 (2012)
11. Morozov, A.A., Sushkova, O.S., Polupanov, A.F.: A translator of Actor Prolog to Java. In: Bassiliades, N., Fodor, P., Giurca, A., Gottlob, G., Kliegr, T., Nalepa, G., Palmirani, M., Paschke, A., Proctor, M., Roman, D., Sadri, F., Stojanovic, N. (eds.) RuleML 2015 DC and Challenge, Berlin, CEUR (2015)
12. Morozov, A.A., Vaish, A., Polupanov, A.F., Antciperov, V.E., Lychkov, I.I., Alfimtsev, A.N., Deviatkov, V.V.: Development of concurrent object-oriented logic programming platform for the intelligent monitoring of anomalous human activities. In: Plantier, G., Schultz, T., Fred, A., Gamboa, H. (eds.) BIOSTEC 2014. CCIS, vol. 511, pp. 82–97. Springer, Cham (2015). doi:10.1007/978-3-319-26129-4_6
13. Morozov, A.A., Polupanov, A.F.: Intelligent visual surveillance logic programming: implementation issues. In: Ströder, T., Swift, T. (eds.) CICLOPS-WLPE 2014. Number AIB-2014-09 in Aachener Informatik Berichte, RWTH Aachen University, pp. 31–45 (2014)
14. Morozov, A.A., Polupanov, A.F.: Development of the logic programming approach to the intelligent monitoring of anomalous human behaviour. In: Paulus, D., Fuchs, C., Droege, D. (eds.) OGRW 2014, pp. 82–85. University of Koblenz-Landau, Koblenz (2015)
15. Morozov, A.A., Sushkova, O.S., Polupanov, A.F.: An approach to the intelligent monitoring of anomalous human behaviour based on the Actor Prolog object-oriented logic language. In: Bassiliades, N., Fodor, P., Giurca, A., Gottlob, G., Kliegr, T., Nalepa, G., Palmirani, M., Paschke, A., Proctor, M., Roman, D., Sadri, F., Stojanovic, N. (eds.) RuleML 2015 DC and Challenge, Berlin, CEUR (2015)
16. Morozov, A.A.: Development of a method for intelligent video monitoring of abnormal behavior of people based on parallel object-oriented logic programming. Pattern Recogn. Image Anal. **25**, 481–492 (2015)
17. Bordini, R.H., Hübner, J.F., Wooldridge, M.: Programming Multi-agent Systems in AgentSpeak Using Jason. Wiley Series in Agent Technology, 8th edn. Wiley, Chichester (2007)
18. Dastani, M.: 2APL: a practical agent programming language. Auton. Agent. Multi-Agent Syst. **16**, 214–248 (2008)
19. Odell, J.: Objects and agents compared. J. Object Technol. **1**, 41–53 (2002)
20. Nierstrasz, O., Dami, L.: Component-oriented software technology. In: Nierstrasz, O., Tsichritzis, D. (eds.) Object-Oriented Software Composition, pp. 3–28. Prentice Hall, Upper Saddle River (1995)
21. Davison, A.: A survey of logic programming-based object oriented languages. Technical report 92/3, Department of Computer Science, University of Melbourne, Melbourne, Australia (1992)
22. Borland International: Turbo Prolog Owner's Handbook (1986)
23. Morozov, A.A., Sushkova, O.S.: The intelligent visual surveillance logic programming Web Site (2016). http://www.fullvision.ru/actor_prolog/