Development of Agent Logic Programming Means for Heterogeneous Multichannel Intelligent Visual Surveillance

Alexei A. Morozov(✉) and Olga S. Sushkova

Kotel'nikov Institute of Radio Engineering and Electronics of RAS, Mokhovaya 11-7, Moscow, Russia
morozov@cplire.ru, o.sushkova@mail.ru
http://www.fullvision.ru

Abstract. Experimental means developed in the Actor Prolog parallel object-oriented logic language for implementation of heterogeneous multichannel intelligent visual surveillance systems are considered. These means are examined by the instance of a logic program for permanent monitoring of people's body parts temperature in the area of visual surveillance. The logic program implements a fusion of heterogeneous data acquired by two devices: (1) 3D coordinates of the human body are measured using a time-of-flight (ToF) camera; (2) 3D coordinates of the human body skeleton are computed on the base of 3D coordinates of the body; (3) a thermal video is acquired using a thermal imaging camera. In the considered example, the thermal video is projected to the 3D surface of the human body; then the temperature of the human body is projected to the vertices and edges of the skeleton. A special logical agent (i.e., the logic program that is written in Actor Prolog) implements these operations in real-time and transfers the data to another logical agent. The latter agent implements a time average of the temperature of the human skeletons and displays colored 3D images of the skeletons; the average temperature of the vertices and edges of the skeletons is depicted by colors. The logic programming means under consideration are developed for the purpose of the implementation of logical analysis of video scene semantics in the intelligent visual surveillance systems.

1 Introduction

In this paper, the basic ideas of using the Actor Prolog object-oriented logic language for the multichannel/multimedia data analysis are described by the example of processing of 3D video data acquired using Kinect 2 (Microsoft, Inc.) and 2D thermal imaging video acquired using the Thermal Expert V1 camera (i3system, Inc.). The distributed logic programming means of Actor Prolog are discussed by the example of two communicating logical agents that analyze 3D video data and implement a fusion of these data with the thermal video.

© Springer Nature Switzerland AG 2018
G. R. Simari et al. (Eds.): IBERAMIA 2018, LNAI 11238, pp. 29–41, 2018.
https://doi.org/10.1007/978-3-030-03928-8_3

The fusion of the thermal imaging video with 3D video data and the data of other kinds is a rapidly developed research area [1,2,15,16]. In this paper, the problem of remote measurement of human body parts in the video surveillance area is considered as an example. In the first section, the architecture and basic principles of the Actor Prolog logic programming system are considered. In the second section, a set of built-in classes of the Actor Prolog language are considered that were developed by the authors for the acquisition and analysis of 3D video data. In the third section, an example of a logical agent that inputs 3D data on the body surface of the people under the video surveillance and implements a fusion of this data with the thermal imaging video is discussed. In the fourth section, the basic principles and means for the communication of the logical agent in the Actor Prolog language are considered.

2 The Architecture of the Actor Prolog Logic Programming System

Actor Prolog is a logic programming language developed in the Kotel'nikov Institute of Radio Engineering and Electronics of Russian Academy of Sciences [3–7]. Actor Prolog was designed initially as an object-oriented and logic language simultaneously, that is, the language implements classes, instances, and inheritance; at the same time, the object-oriented logic programs have model-theory semantics. The Actor Prolog language supports means for definition of data types (so-called domains), the determinancy of predicates, and the direction of data transfer in the subroutine arguments [14]. These means are vital for the industrial applications of the logic programming because the experience demonstrated that it is very difficult to support and debug big and complex logic programs without these means. Actor Prolog is a parallel language; there are syntax means in the language that support creation and control of communicating parallel processes. These syntax means of the language provide the model-theory semantics of the logic programs as well, but only when certain restrictions are imposed on the syntax and structure of the programs [6].

A distinctive feature of the Actor Prolog logic programming system is in that the logic programs are translated in Java code and executed by the standard Java virtual machine [9,11]. This scheme of logic program execution was developed, mainly, to ensure the stability of the programs and prevent possible problems with the memory management. Another important feature of this scheme is in that it ensures high extensibility of the logic language; one can easily add necessary built-in classes to the language.

One can add a new built-in class to the Actor Prolog language in the following way. During the translation of the logic program, it is converted to the set of Java classes. There are special syntax means in the language to declare some automatically generated Java classes as descendants of external Java classes that were created manually by the programmer. Thus, it is enough to implement a new built-in class in Java and link it with the logic program in the course of translation to make this class the built-in class of Actor Prolog. Currently, a set

of built-in classes of Actor Prolog are implemented totally in pure Java; other built-in classes are Java interfaces with open source libraries implemented in C++. The examples of the former classes are: the *Database* class that implements a simple data management system; the *File* class that supports reading and writing files; the *WebResource* class that implements data acquisition from the Web. The examples of the latter classes are: the *FFmpeg* class that links Actor Prolog with the FFmpeg open source library for video reading and writing; the *Java3D* class that implements 3D graphics; the *Webcam* class that supports video data acquisition. The authors consider the translation to Java as an architectural solution that helps to develop and debug rapidly new built-in classes. The speeding-up of the software life cycle is caused by the fact that the Java language, in comparison with the C++ language, prevents the appearance of the bugs linked with the incorrect memory access and out-of-range array access that can be very difficult for detection.

3 Built-In Classes Supporting the Intelligent Visual Surveillance

A set of built-in classes for 2D and 3D video acquisition and analysis is implemented in the Actor Prolog logic programming system. These built-in classes were developed mainly in the course of experimenting with the methods of intelligent video surveillance.

In this paper, new means of the Actor Prolog language are described that were developed for 3D data acquisition and analysis using the Kinect 2 device. These means and specialized built-in classes of Actor Prolog were created for the experimenting with the 3D intelligent visual surveillance [12,13]. The developed means are based on the same ideas as the means for the 2D video analysis:

1. The low-level and high-level video processing stages are separated.
2. The low-level video processing stage is implemented in special built-in classes that encapsulate all intermediate data matrices.
3. The high-level video processing stage is implemented by the logical rules; the data is processed in the form of graphs, lists, and other terms of the logic language.

The data processing scheme was adapted to the following properties of 3D video data:

1. The data are heterogeneous (multimodal). For instance, the Kinect 2 device provides several data streams simultaneously; there are: the frames describing 3D point clouds; the infrared imaging frames; the conventional colored (RGB) frames; the frames that describe coordinates of human skeletons; etc.
2. The size of 3D video data is usually huge; a typical personal computer is not powerful enough to store in real time all raw data of Kinect 2 to the hard disk. Thus, a preferable scheme of 3D data processing includes preliminary real-time analysis of the data and storing/networking the intermediate results of the analysis.

There are two built-in classes in Actor Prolog that support 3D video acquisition and analysis: *Kinect* and *KinectBuffer*. The former class implements communication between the logic program and the Kinect 2 device. The latter class implements low-level analysis as well as reading and writing 3D data files. The definitions of these classes including the definitions of data types (domains) and predicates are placed in the "Morozov/Kinect" package of Actor Prolog.

The *KinectBuffer* class is the most important element in the 3D data processing scheme. The instance of the *KinectBuffer* class can be used in the following modes: data acquisition from Kinect 2; reading data from the file; playing 3D video file; writing data to the file. The playing-3D-video-file and reading-from-the-file modes differ in that in the former mode the *KinectBuffer* class reads data and transfers them to the logic program in real-time; in the reading-from-the-file mode, the programmer has to control reading of every frame of the video.

The *operating_mode* attribute of the *KinectBuffer* class is to be used to select the operating mode of the instance of the class: *LISTENING*, *READING*, *PLAYING*, or *RECORDING* correspondingly. The *KinectBuffer* class can be used alone to read/write 3D videos; however one has to use it in connection with the *Kinect* class to acquire data from the Kinect 2 device. In this mode, one has to create an instance of the *Kinect* class and transmit it to the constructor of the *KinectBuffer* class instance; the *input_device* attribute is to be used as the argument of the constructor. An example of the logic program that reads and processes 3D video data from the file is considered in the next section.

4 Acquisition and Fusion of 3D and Thermal Imaging Video Data

Let us consider an example of the logic program that reads and implements a simple analysis of 3D video data that were acquired using ToF camera of the Kinect 2 device. Fragments of source code written in Actor Prolog will be demonstrated below with comments; of course, the source code is reduced, because the purpose of the example is just to describe the scheme of the data processing using the *Kinect* and *KinectBuffer* classes.

Let us define the *3DVideoSupplier* class that is a descendant of the *Kinect Buffer* class. The *operating_mode* attribute has the *PLAYING* value that indicates that the data are to be read from the "My3DVideo" file in the playing-3D-video-file mode.

```
class '3DVideoSupplier' (specialized 'KinectBuffer'):
name                 = "My3DVideo";
operating_mode       = 'PLAYING';
```

In the course of the creation of the *3DVideoSupplier* class instance, it downloads a lookup table from the "MyLookUpTable.txt" file. This lookup table establishes the correspondence between the 3D coordinates measured by the ToF

camera and 2D coordinates on the 2D thermal imaging video. After that, the reading from the file is activated using the *start* predicate of the $KinectBuffer$ class.

```
goal:-!,
     set_lookup_table("MyLookUpTable.txt"),
     start.
```

The $KinectBuffer$ class supports 2D and 3D lookup tables. It is supposed that the lookup table is computed and stored in the text file in advance during the calibration of the video acquisition system. The 2D lookup table is a matrix K of the same size as the Kinect 2 infrared video frame. Each cell (i, j) of the matrix contains coordinates x and y on an image T; in the example under consideration, T is the thermal image. Thus, the T image is to be projected to 3D surfaces investigated using the ToF camera. The 3D lookup table is also a matrix, but the cell of this matrix contains quadratic polynomial coefficients that are necessary for computation of the coordinates on the T image, but not the (x, y) coordinates themselves. Each cell (i, j) of the matrix contains six coefficients p_1, p_2, p_3, q_1, q_2, and q_3. The coordinates on the T image are computed using the quadratic polynomial depending on the inverse value of the distance $d(i, j)$ in meters between the ToF camera and the surface of the object to be investigated:

$$x = p_1(1/d)^2 + p_2(1/d) + p_3 \tag{1}$$

$$y = q_1(1/d)^2 + q_2(1/d) + q_3 \tag{2}$$

In the example, the thermal image is projected to the 3D surface during the processing of every frame of 3D video. The $frame_obtained$ predicate is invoked automatically in the instance of the $KinectBuffer$ class when a new frame of 3D video is read from the file. The programmer informs the $KinectBuffer$ class that s/he is going to process this frame using the *commit* predicate. After that, all the predicates of the $KinectBuffer$ class operate with the content of this particular flame until the *commit* predicate is called again. The logic program gets the $Time1$ time of the frame in milliseconds using the $get_recent_frame_time$ predicate. Then the number of corresponding thermal imaging frame is calculated using this information. The $texture_time_shift$ attribute contains a value of the temporal shift between the 3D and thermal videos. The $texture_frame_rate$ attribute contains the frame rate of the thermal imaging video.

```
frame_obtained:-
     commit,!,
     get_recent_frame_time(Time1),
     Time2== Time1 - texture_time_shift,
     FileNumber== texture_frame_rate * Time2 / 1000,
```

Suppose the thermal imaging video is converted to the separate frames. The $frame_obtained$ predicate computes the name of the corresponding JPEG file using the number of the frame. Then it uses the *load* predicate to read the

frame and stores it to the image instance of the *BufferedImage* built-in class. The *get_recent_scene* predicate of the *KinectBuffer* class is called; this predicate creates a 3D surface on the base of the ToF camera data and projects given texture to this surface. The texture is transferred to the predicate by the second argument *image* that contains an instance of the *BufferedImage* class. The lookup table loaded above is used for the implementation of the texture projection. The created 3D surface is returned from the *get_recent_scene* predicate via the first argument. This argument has to contain an instance of the *BufferedScene* built-in class. The *BufferedScene* built-in class implements storing and transferring 3D data. In particular, the content of the *BufferedScene* class instance can be inserted into the 3D scene displayed by the means of the Java3D graphics library. In the example under consideration, the *set_node* predicate is used for this purpose that replaces the "MyLabel" node of the 3D image created using the *graphics_window* instance of the *Java3D* built-in class.

```
ImageToBeLoaded == text?format(
     "%08d.jpeg",?round(FileNumber)),
image ? load(ImageToBeLoaded),
get_recent_scene(buffer3D,image),
graphics_window ? set_node(
     "MyLabel",
     'BranchGroup'({
          label: "MyLabel",
          allowDetach: 'yes',
          compile: 'yes',
          branches: [buffer3D]
     })),
```

The *get_recent_mapping* predicate of the *KinectBuffer* built-in class operates approximately in the same way as the *get_recent_scene* predicate. The difference is in that it does not create a 3D surface and the results of the projection of the texture to the 3D surface are returned in a form of a convenient 2D image. In the example, the *get_recent_mapping* predicate stores the created image to the *buffer2D* instance of the *BufferedImage* built-in class. The *get_skeletons* predicate of the *KinectBuffer* class returns a list of the skeletons detected in the current frame. The skeletons are graphs that contain information about coordinates of the human body, head, arms, and legs. In the logic program, the graphs are described using the standard simple and compound terms: lists, structures, underdetermined sets, symbols, and numbers [12,13]. In the example, the skeletons and thermal images are transferred to another logical agent that implements further analysis and fusion of the data. The routine of the data transfer between the logical agents will be considered in the next section. Note that the image to be transferred from the *buffer2D* instance of the *BufferedImage* class is converted to the term of the *BINARY* type. The *get_binary* predicate of the *BufferedImage* class is used for this purpose.

Fig. 1. An example of a logical agent that collects and transfers heterogeneous multi-channel data (3D video data and thermal imaging data).

```
get_recent_mapping(buffer2D,image),
get_skeletons(Skeletons),
communicator ? notify_all_consumers(
     Skeletons,buffer2D?get_binary()).
```

The results of the fusion of 3D and thermal imaging data implemented by the logical agent under consideration are demonstrated in the Fig. 1.

In the next section, a scheme of communication between the logic programs (the logical agents) based on the *Database* and *DataStore* built-in classes and the mechanism of the remote predicate calls are discussed.

5 A Link Startup and Communication Between the Logical Agents

The remote predicate calls are a special feature of the Actor Prolog language that was developed to support distributed/decentralized logic programming. The idea of this mechanism is in that any object of the logic program (the logical agent) can be transferred to another logical agent; after that, the new owner of the object can invoke remotely and asynchronously the predicates of the object [14]. Note that in terms of Actor Prolog the object is a synonym of the world and the instance of a class. The complication of the development of this mechanism was in that Actor Prolog has a strong type system and, therefore, the type system of the language did not provide a possibility for the agents to link and communicate dynamically without a preliminary exchange of the information about the types of the data to be transferred. Usually, the languages with strong type systems require an exchange of data type definitions on the stage of compilation of the agents, but in the Actor Prolog language, another solution was elaborated. In the distributed version of Actor Prolog, a combined type system was developed,

that is, the strong type system was partially softened in the case when the inter-agent data exchange is performed. To be more precise, the types of the arguments in the remote predicate calls are compared by structure, but not by names; moreover, the check of that the external object implements a required predicate is postponed until this predicate call is to be actually performed. In all other cases, the standard static type check is implemented in the language. The combined type system unites the advantages of the strong type system that is used for generation of reliable and fast executable code and the possibility of the dynamic type check during the inter-agent data exchange.

An instance of a class of the logical agent is to be transferred to other agents somehow to establish a connection between the agents. The instance of the class can be transferred via an operating system file, a shared database, or just in a text form by E-mail. In the example under consideration, the built-in database management system of Actor Prolog is used for this purpose. This system is implemented by the *Database* and *DataStore* built-in classes of the language.

Let us define the *Main* class that will implement two roles in the logic program: the execution of the logic program begins with the creation of an instance of the *Main* class in accordance with the definition of the language; the instance of the class is to be transferred to another agent to establish a link and for the communication in the example under consideration.

The *Main* class contains several slots (that is, the class instance variables). The *datastore* slot contains an instance of the *DataStore* built-in class. The *database* slot contains an instance of the *3DDataSources* class that is a descendant of the *Database* built-in class. The *video_supplier* slot contains an instance of the *3DVideoSupplier* class that was considered in the previous section. The *consumers* slot contains an instance of the *ConsumersList* class that is a descendant of the *Database* built-in class; the *consumers* database keeps a list of external logical agents that requested information from the agent under consideration.

```
class 'Main' (specialized 'Alpha'):
datastore = ('DataStore',
               name="AgentBlackboard.db",
               sharing_mode='shared_access',
               access_mode='modifying');
database  = ('3DDataSources',
               place= shared(
                    datastore,
                    "3DDataSources"));
video_supplier = ('3DVideoSupplier',
               communicator=self);
consumers      = ('ConsumersList');
```

The *Database* built-in class implements a simple database management system that provides storing and searching the data of arbitrary structure; the operations of this kind are standard for the Prolog-like logic languages. One

might say that the convenient relational databases are a special case of the Prolog databases when the database is used only for storing the data of the structure type, that is, the records with a name and a list of arguments. Note that the *Database* class is destined for storing data in the main memory of the computer, that is, for the management of the temporary data. The temporary data can be stored to the file or loaded from the file if necessary.

A database management mechanism of a more high level is to be used to control data that are shared between several logic programs. This mechanism is implemented in the *DataStore* built-in class. The *DataStore* class can coordinate and control the operation of one or several instances of the *Database* class. For instance, one can read from or write to the file the content of several instances of the *Database* class at once. Another useful mechanism of the *DataStore* class is supporting shared data access, that is, it can link the instances of the *Database* class with the operating system files and automatically transfer the updates of the data in the memory of one logic program to the memory of other logic programs. The data integrity is guaranteed by the standard mechanism of transactions. These instruments of the *DataStore* class are used in the example under consideration.

In the code above, the constructor of the *DataStore* class instance accepts the following input arguments: the *name* attribute that contains the name of the "AgentBlackboard.db" file that is to be used for the shared data storage; the *sharing_mode* attribute that assigns the shared mode of the data access (the *shared_access* mode); the *access_mode* attribute that indicates that the logic program demands the privileges for shared data modification (the *modifying* mode).

The constructor of the 3*DDataSources* class instance accepts the *place* argument that contains the *shared*(*datastore*, "3*DDataSources*") structure with two internal arguments. This argument indicates that the instance of the 3*DData Sources* database will operate under the control of the *DataStore* class and the content of the database has the "3DDataSources" unique identifier in the namespace of the *DataStore* class instance. Using the analogy with the relational databases, "3DDataSources" is the name of a generalized relational table in the "AgentBlackboard.db" shared database.

In the course of the creation of the *Main* class instance, a sequence of operations on the shared data will be performed. The *open* predicate initiates the access to the "AgentBlackboard.db" shared data file. After that, a transaction will be opened with the data modification privileges. All the records in the "3DDataSources" database will be deleted and the *Main* class instance will store itself in the database. Then the transaction will be completed by the *end_transaction* predicate and the access to the "AgentBlackboard.db" database will be ended by the *close* predicate.

```
goal:-
     datastore ? open,
     database ? begin_transaction('modifying'),!,
     database ? retract_all(),
```

```
     database ? insert(self),
     database ? end_transaction,
     datastore ? close.
```

The instance of the *Main* class becomes available for other logical agents after the storing to the shaded database. In particular, an external agent can read this instance from the database and invoke the *register_consumer* predicate in the world to receive the information about the temperature of the people in the scope of the visual surveillance system. The external agent has to send himself as the argument of the *register_consumer* predicate. The *register_consumer* predicate of the former agent will store it to the *consumers* internal database.

```
register_consumer(ExternalAgent):-
     consumers ? insert(ExternalAgent).
```

During each call of the *frame_obtained* predicate of the class *3DVideo Supplier*, the *notify_all_consumers* predicate is called. This predicate uses the search with backtracking to extract one-by-one the receivers from the *consumers* database and send them the data set.

```
notify_all_consumers(Skeletons,Image):-
     consumers ? find(ExternalAgent),
          notify_consumer(
               ExternalAgent,Skeletons,Image),
          fail.
notify_all_consumers(_,_).
```

The *notify_consumer* predicate implements the remote call of the predicate *new_frame* in the *ExternalAgent* external world.

```
notify_consumer(ExternalAgent,Skeletons,Image):-
     [ExternalAgent] [<<] new_frame(Skeletons,Image).
```

The syntax notation of this clause has the following semantics. The << infix indicates that so-called informational direct message is to be used for the data transfer. The informational direct message is a kind of asynchronous messages supported by the Actor Prolog language [6]. The square brackets in the [<<] expression indicate that this message must not be buffered, that is, the message is to be discarded if the receiver is not processed it before the receiving the next message. The square brackets in the [*ExternalAgent*] expression indicate that the remote call of the *new_frame* predicate is to be performed immediately during the execution of the command under consideration; otherwise the remote call will be suspended and afterward canceled during the backtracking of the *notify_all_consumers* predicate.

The receiving logical agent will accept and process the message *new_frame* (*Skeletons*, *Image*). In the example, the receiving agent has to analyze the *Skeletons* graph describing the skeletons of the people observed by the intelligent video surveillance system. The vertices and edges of the graph with the

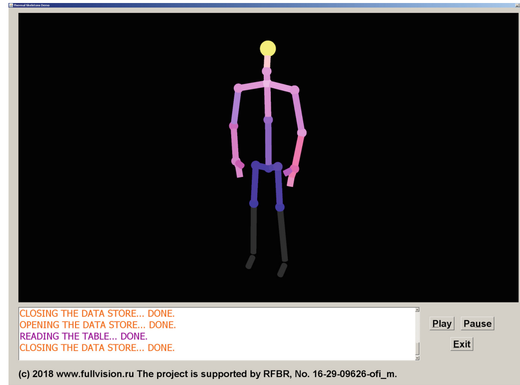

Fig. 2. A logical agent that estimates the average temperature of the human body parts during the unrestricted movements of people in the video scene.

$TRACKED$ status (that means that the ToF camera observes them directly) will be selected and compared with the $Image$ thermal imaging frame. The intensities of the pixels at the thermal image that correspond to the coordinates of the graph edges are averaged. The intensities of vertices and average intensities of edges are stored in a special table. In the course of the processing of new frames, a time average of the intensities of vertices and edges stored in the table will be implemented. The purpose of this processing is in that one estimates the average temperature of the human body parts during the unrestricted movements of the people in the video scene (see the Fig. 2).

The principles of analysis and fusion of heterogeneous multichannel video data by the means of the object-oriented logic programming were discussed by the example. A set of built-in classes of the Actor Prolog language for the database management as well as acquisition and distributed processing of 2D and 3D video data were considered. The complete text of the example considered in this paper is published in the installation package of the Actor Prolog that is freely available in the Web Site [10]. The source codes of the built-in classes considered in the paper are available on GitHub [8].

6 Conclusions

In the Actor Prolog project, mean and tools for 2D and 3D video data acquisition and processing are developed and implemented including the means for the fusion of 3D data collected using a ToF camera and a thermal imaging camera. In the authors' opinion, the main result of the project is in that the logic language is adopted for the effective processing of the big arrays of heterogeneous data. This is provided thanks to the object-oriented architecture of the Actor Prolog logic programming system that supplies the encapsulation of the big data arrays in the instances of specialized built-in classes. The developed logical means open

new prospects in the area of intelligent visual surveillance, namely, they enable to implement semantic analysis of the video scenes, that is, conduct logical inference on the base of heterogeneous data describing the context of the human actions, objects, and events observed by the intelligent video surveillance system.

Acknowledgement. This research was supported by the Russian Foundation for Basic Research (grant number 16-29-09626-ofi-m).

References

1. Gade, R., Moeslund, T.B.: Thermal cameras and applications: a survey. Mach. Vis. Appl. **25**(1), 245–262 (2014)
2. Han, S., Gu, X., Gu, X.: An accurate calibration method of a multi camera system. In: Fei, M., Ma, S., Li, X., Sun, X., Jia, L., Su, Z. (eds.) LSMS/ICSEE -2017. CCIS, vol. 761, pp. 491–501. Springer, Singapore (2017). https://doi.org/10.1007/978-981-10-6370-1_49
3. Morozov, A., Obukhov, Y.: An approach to logic programming of intelligent agents for searching and recognizing information on the Internet. Pattern Recogn. Image Anal. **11**(3), 570–582 (2001)
4. Morozov, A.A.: Actor Prolog. Programmirovanie **5**, 66–78 (1994). in Russian
5. Morozov, A.A.: Actor Prolog: an object-oriented language with the classical declarative semantics. In: Sagonas, K., Tarau, P. (eds.) IDL 1999, pp. 39–53. France, Paris (1999)
6. Morozov, A.A.: Logic object-oriented model of asynchronous concurrent computations. Pattern Recogn. Image Anal. **13**(4), 640–649 (2003)
7. Morozov, A.A.: Operational approach to the modified reasoning, based on the concept of repeated proving and logical actors. In: Salvador Abreu, V.S.C. (ed.) CICLOPS 2007, Porto, Portugal, pp. 1–15 (2007)
8. Morozov, A.A.: A GitHub repository containing source codes of Actor Prolog built-in classes (2018). https://github.com/Morozov2012/actor-prolog-java-library
9. Morozov, A.A., Polupanov, A.F.: Intelligent visual surveillance logic programming: implementation issues. In: Ströder, T., Swift, T. (eds.) CICLOPS-WLPE 2014, pp. 31–45. No. AIB-2014-09 in Aachener Informatik Berichte, RWTH Aachen University, June 2014
10. Morozov, A.A., Sushkova, O.S.: The intelligent visual surveillance logic programming Web Site (2018). http://www.fullvision.ru
11. Morozov, A.A., Sushkova, O.S., Polupanov, A.F.: A translator of Actor Prolog to Java. In: Bassiliades, N., et al. (eds.) RuleML 2015 DC and Challenge. CEUR, Berlin (2015)
12. Morozov, A.A., Sushkova, O.S., Polupanov, A.F.: Object-oriented logic programming of 3D intelligent video surveillance: the problem statement. In: ISIE, pp. 1631–1636. IEEE Xplore Digital Library, Washington (2017)
13. Morozov, A.A., Sushkova, O.S., Polupanov, A.F.: Object-oriented logic programming of 3D intelligent video surveillance systems: the problem statement. RENSIT **2**(9), 205–214 (2017)
14. Morozov, A.A., Sushkova, O.S., Polupanov, A.F.: Towards the distributed logic programming of intelligent visual surveillance applications. In: Pichardo-Lagunas, O., Miranda-Jiménez, S. (eds.) MICAI 2016. LNCS (LNAI), vol. 10062, pp. 42–53. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-62428-0_4

15. Nakagawa, W., et al.: Visualization of temperature change using RGB-D camera and thermal camera. In: Agapito, L., Bronstein, M.M., Rother, C. (eds.) ECCV 2014. LNCS, vol. 8925, pp. 386–400. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-16178-5_27
16. Rangel, J., Soldan, S., Kroll, A.: 3D thermal imaging: fusion of thermography and depth cameras. In: International Conference on Quantitative InfraRed Thermography (2014)