Development of a Method for Intelligent Video Monitoring of Abnormal Behavior of People Based on Parallel Object-Oriented Logic Programming[1]

A. A. Morozov

Kotel'nikov Institute of Radio Engineering and Electronics, Russian Academy of Sciences, ul. Mokhovaya 11, str. 7, Moscow, 125009 Russia

e-mail: morozov@cplire.ru

Abstract—A method for intelligent video monitoring of abnormal behavior of people is considered that is based on parallel object-oriented logic programming. The main idea of the method consists in the description of sought scenarios of abnormal behavior of people followed by analysis of video images by logic programming means (using first-order predicate logic). The distinctive features of this method are the use of the Actor Prolog object-oriented logic language and the translation of logic programs of intelligent video surveillance into the Java language. In this case, the object-oriented means of the logic language allow one to split a program into interacting parallel processes that implement various stages of video image processing and scene analysis, while translation into the Java language guarantees the necessary reliability, portability, and openness of the software for intelligent video monitoring, including the possibility of using modern libraries of low-level image processing.

Keywords: intelligent video surveillance, parallel object-oriented logic programming, detection of abnormal behavior, abnormal behavior of people, Actor Prolog, complex event recognition, machine vision, technical vision, video image analysis, Prolog-to-Java translation.

DOI: 10.1134/S1054661815030153

INTRODUCTION

Detection of abnormal behavior of people is a topical and rapidly developing field of research that is directly related to intelligent video surveillance, life insurance, and the fight against terrorism [1–3]. In recent years, logic programming of intelligent video surveillance has been considered as one of the most promising areas of research in the field of recognition of abnormal behavior [4–9].

The idea of this approach is that the description and subsequent analysis of the scenarios of abnormal behavior of people and technical objects use a logic language; in this case, logical rules describe temporal and spatial relations between the objects to be analyzed and elementary events, as well as the properties of objects that are related to the problem of recognition of abnormal behavior. Typically, one distinguishes low-level recognition problems (e.g., background subtraction of a video image, detection of people and cars in a video image, construction of the trajectories of moving objects, recognition of the pose and certain parts of a human body, evaluation of its velocity, detection of abrupt movements, etc.) and high-level recognition problems (e.g., recognition of a fight, armed assault, theft, unattended objects, and others). High-level recognition problems are solved with the use of logical means; in this case, the input data are given by the results of low-level recognition obtained with the use of standard methods of video image analysis and lower level programming languages.

Today, there are several research projects that are using the idea of logic programming of intelligent video surveillance. In particular, the research group of Artikis [4, 5] is developing an approach based on event calculus implemented with the use of a logic language. Within this approach, so-called "long-term types of activities" (meeting, fighting, etc.) are recognized as combinations of "short-term types" (walking, running, being inactive, etc.). These authors call attention to the problem of uncertainty in the recognition of different types of human activity; the ProbLog system of probabilistic logic programming is applied to solve this problem. Another approach to solving the uncertainty problem in the recognition of behavior, which is based on the application of predicate logic extended by the so-called bilattices, was developed by Shet et al. [6]. Earlier, the same author developed a VidMAP system of intelligent video surveillance [7], which combines the Prolog programming language with low-level algorithms for real-time analysis of video images. Another intelligent video surveillance system, VERSA, which is based on the SWI–Prolog logic language, was developed by O'Hara [8]. According to this scientist, it has more powerful means of temporal and spatial reasoning than does the VidMAP system. Machot et al. [9] have described a recog-

[1] This paper uses the materials of the report submitted at the 11th International Conference "Pattern Recognition and Image Analysis: New Information Technologies," Samara, Russia, September 23–28, 2013.

Received January 30, 2014

Fig. 1. A situation like a street incident (fight between two people) comes into the field of view of the intelligent video surveillance program. Heavy solid lines show the trajectories of people, and thin lines and circles indicate connected graphs of motions of blobs.

nition system for complex audio–video events that is based on Answer Set Programming. According to these scientists, the recognition method proposed can be implemented on a special chip.

Use of the object-oriented logic language Actor Prolog [10–18], as well as the translation of logic programs of intelligent video surveillance into the Java language [16], are distinctive features of the method for recognition of abnormal behavior considered in this paper. In this case, the object-oriented means of the logic language allow one to split a program into interacting parallel processes that implement various stages of image processing and scene analysis, while translation into the Java language provides the necessary reliability, portability, and openness of the intelligent video monitoring software, including the possibility of using modern low-level image-processing libraries.

The article deals with the basic ideas and principles that underlie the method developed for detecting abnormal behavior. The first section considers the structure of the simplest intelligent video surveillance program and the implementation of the main stages of video image analysis by the means of parallel object-oriented logic language. The second section is devoted to a detailed description of the built-in means of Actor Prolog that implement a low-level analysis of video images. In the third section, we discuss the description of scenarios of abnormal behavior of people by the means of the logic language and the issues of high-level analysis of video images.

1. MAIN STAGES OF VIDEO IMAGE ANALYSIS

As a rule, video image analysis is naturally divided into individual stages that can be implemented in the form of interacting parallel processes. The application of parallel programming is very convenient from the point of view of structuring the text of a program and becomes absolutely indispensable when analyzing video images arriving in real time, because different stages of analysis have different priorities. Some operations (for example, background subtraction, extraction of blobs, tracing of objects, etc.) are crucial because the omission of these operations may lead to a loss of video capture of objects. Other operations (analysis of the interaction graphs of objects, visualization of the results of analysis, etc.) can be suspended to save computational resources. Unfortunately, most logic languages, including the first logic language, Prolog, have been initially created as sequential programming languages; parallel calculations have been provided for neither at the level of the mathematical theory nor at the level of computer implementation. Parallel logic languages have long ceased to be exotic; nevertheless, existing projects of logic programming of intelligent video surveillance so far have been based on sequential logic languages.

The method of logic programming of intelligent video monitoring considered in this paper is based on the use of object-oriented means of the Actor Prolog logic language. In Actor Prolog, the text of a logic program consists of separate classes [10], and parallel processes represent a kind of instances of classes [12]. We will consider the application of these means of the logic language by an example of the analysis of one standard video clip from the CAVIAR collection [19].

Below we sketch a logic program that recognizes a street incident (Fig. 1).

The program creates two parallel processes. The first process reads a sequence of JPEG images that mimics a real-time input of a video and carries out a low-level analysis, namely, the subtraction of background, discrimination of blobs, tracing of objects, and the determination of the points of interaction of blobs. All these operations are implemented by means of a special built-in class '*ImageSubtractor*' of Actor Prolog, which will be considered in the following section of the article. The second process analyzes the information prepared in the first process and displays the results of video surveillance on the screen. In general, these two operations could also be implemented in the form of separate processes, but we will not do this for the sake of simplicity of exposition.

To create a program, we will use the following predefined packages of Actor Prolog: *Java2D*, which implements two-dimensional graphics, and *Vision*, which is developed within our project and implements low-level operations of video processing.

```
import .. from "morozov/Java2D";
import .. from "morozov/Vision".
```

The main class '*Main*' does not contain any logical rules; it is necessary only to create and interconnect auxiliary instances of classes. The '*Main*' class contains the *data_directory* and *target_directory* slots, which contain the names of directory and subdirectory

where JPEG images are stored; the *sampling_rate* slot, which contains the frame rate of a video clip; the *stage_one* and *stage_two* slots, which contain processes that implement the first and second steps of video processing, respectively; and the *low_level_analyzer* slot, which contains an instance of the built-in class '*ImageSubtractor*.' Note that the constructors of the instances of the '*ImagePreprocessor*' and '*ImageAnalyser*' classes are enclosed in double parentheses; according to the syntax of Actor Prolog, this means that these instances of classes are processes that are executed in parallel with respect to other processes of the program (that is, with respect to each other, as well as with respect to the instance of the `'Main'` class). Note also that an instance of the *'ImageSubtractor*' class is transferred to the *stage_one* and *stage_two* processes as an argument; moreover, the *stage_two* instance of the '*ImageAnalyser*' class is transferred to the *stage_one* process in order that the former could send messages to it. Here a part of arguments of an instance of the '*ImageSubtractor'* class are omitted and will be considered in the second section of the paper.

```
class 'Main' (specialized 'Alpha'):
constant:
      data_directory       = "data";
      target_directory     = "Fight_RunAway1";
      sampling_rate        = 25.0;
      stage_one            = (('ImagePreprocessor',
                              data_directory,
                              target_directory,
                              sampling_rate,
                              low_level_analyzer,
                              stage_two));
      stage_two            = (('ImageAnalyser',
                              low_level_analyzer,
                              sampling_rate));
internal:
      low_level_analyzer   = ('ImageSubtractor',
                              extract_blobs= 'yes',
                              track_blobs= 'yes',
                              …);
[
]
```

The '*ImagePreprocessor*' class is a descendant of the built-in class '*Timer*.' The meaning of this inheritance, as well as the logical rules defined within the '*ImagePreprocessor*' class, will be considered in the following section (here the rules of a class are omitted and replaced by an ellipsis). The class '*ImagePreprocessor*' uses the *data_directory*, *target_directory*, *sampling_rate*, *stage_two*, and *low_level_analyzer* slots, the meaning of which has been explained above, as well as auxiliary slots that contain instances of classes that are internal with respect to an instance of the class '*ImagePreprocessor*.' In particular, the slot *subtractor* contains an instance of the built-in class '*SynchronizedImageSubtractor*,' to which, in turn, an instance of the *low_level_analyzer* class is transferred as an argument. In addition, the *text* slot, which contains an instance of the built-in class '*Text*,' the *image* slot, which contains an instance of the built-in class '*BufferedImage*,' and the *state* slot, which contains an instance of the '*ProgramState*' class, are defined. The purpose of these slots will be explained below; however, the use of the '*SynchronizedImageSubtractor*' class in a program is directly related to its structure and therefore should be explained in greater detail here.

Owing to their mathematical origin, logic languages are poorly suited for storing large volumes of bitmap data, which are exemplified by photos and video images. The point is that data presentation in logic languages involves terms, that is, in fact, hierarchical structures, symbols, numbers, and text strings. Logic languages contain no data arrays in the form in which they are used in imperative languages, and the introduction of syntactic means similar to arrays into a logic language is a separate mathematical (and engineering) problem. Within our method of video image analysis, video data are stored in the built-in class '*ImageSubtractor*' of Actor Prolog, and all operations with these data are carried out by the standard procedures of this class. That is precisely why an instance of the '*ImageSubtractor*' class is transferred and used in both processes that implement video image processing. However, such a use of an instance of the '*ImageSubtractor*' class may conflict with the idea of Actor Prolog, which forbids a direct call of the procedures of

one process from another process (only a transfer of asynchronous messages between processes is allowed) [12]. The built-in class '*SynchronizedImageSubtractor*' is applied precisely to prevent such collisions. The instances of this class are used as a wrapper for the instances of the '*ImageSubtractor*' class.

In this example, an instance of the class '*SynchronizedImageSubtractor*' is internal with respect to an instance of the '*ImagePreprocessor*' class; therefore, procedures of the class '*SynchronizedImageSubtractor*' can be called from the '*ImagePreprocessor*' class without any restrictions. In this case, the '*SynchronizedImageSubtractor*' class implements the corresponding operations over the data of the class '*ImageSubtractor*' and also provides synchronization and correct access to data from several parallel processes.

```
class 'ImagePreprocessor' (specialized 'Timer'):
constant:
    data_directory;
    target_directory;
    sampling_rate;
    stage_two;
    low_level_analyzer;
internal:
    subtractor          = ('SynchronizedImageSubtractor';
                               image_subtractor = low_level_analyzer);
    text                = ('Text');
    image               = ('BufferedImage');
    state               = ('ProgramState');
[
CLAUSES:
…
]
```

The class '*ImageAnalyser*' uses the *sampling_rate* and *low_level_analyzer* slots considered above, as well as auxiliary slots that contain instances of classes that are internal with respect to the '*ImageAnalyser*' class. The slot *subtractor* contains an instance of the built-in class '*SynchronizedImageSubtractor*,' which is used in exactly the same way as in the '*ImagePreprocessor*' class. The slot *graphic_window* contains an instance of the built-in class '*Canvas2D*,' the slot *image* contains an instance of the built-in class '*BufferedImage*,' and the slot *timer* contains an instance of the built-in class '*Timer*.'

```
class 'ImageAnalyser' (specialized 'Alpha'):
constant:
    sampling_rate;
    low_level_analyzer;
internal:
    subtractor          = ('SynchronizedImageSubtractor',
                              image_subtractor= low_level_analyzer);
    graphic_window      = ('Canvas2D');
    image               = ('BufferedImage');
    timer               = ('Timer');
[
CLAUSES:
goal:-!,
    timer ? set_priority('minimal'),
    graphic_window ? show.
…
]
```

According to the semantics of Actor Prolog, after creation of an instance of the '*ImageAnalyser*' class, a *goal* procedure is automatically called in this class, which sets reduced priority to the process considered (a *set_priority* predicate of the built-in class '*Timer*' with argument '*minimal*' is called) and creates a graphic win-

dow for outputting the results of intelligent video monitoring (a *show* predicate of the built-in class '*Canvas2D*' is called). The purpose of other slots of the class, as well as other rules defined in the class '*ImageAnalyser*,' will be considered in Section 3 of the article.

Thus, with regard to the definitions given above, the '*ImagePreprocessor*' process is responsible for the real-time input of video information and low-level processing. The video information inputted and the results of low-level processing are accumulated in the class '*ImageSubtractor*.' The '*ImageAnalyser*' process has lower priority; as far as possible, it receives and analyzes the data accumulated in an instance of the '*ImageSubtractor*' class and displays the results of logical analysis on the screen.

2. LOW-LEVEL ANALYSIS OF VIDEO IMAGES

A widespread approach to the development and debugging of methods of logical analysis of video images is to use test video clips, low-level processing of which (for example, the recognition and tracing of objects) is carried out manually. Based on our experience, we assume that such an approach is erroneous, because the capabilities and the quality of work of high-level methods of analysis of video images essentially depend on the quality of preliminary processing that can be guaranteed by the low-level methods chosen. A characteristic example of this relationship between high-level and low-level methods is given by the problem of recognition of so-called "abrupt movements," which is of great practical importance for recognizing "fight" and "street offense" situations. Formally, an "abrupt movement" can mean a movement of a human being or part of his body such that the second derivative of its coordinates exceeds some preset threshold. However, in practice, calculation of the second derivative with acceptable accuracy requires knowledge of the precise coordinates of a moving object, which is associated with several serious problems of low-level processing.

1. Generally, it is rather difficult to define even the exact physical coordinates of a human being in a video image. Usually, to solve this problem, one applies the so-called ground plane assumption, which allows one to evaluate the exact coordinates of feet on the basis of several (at least four) reference points on the image. The consideration of a complex relief of the floor and steps, as well as partial overlapping of objects, makes the problem an order of magnitude more difficult.

2. Even the evaluation of the first derivative of the physical coordinates of a moving human being represents a hard problem, because, while moving, a human being appears in different illumination conditions, casts shadows, and is partially or completely overlapped by other objects. As a result, the silhouette of a human being changes unpredictably, and the coordinates of his trajectory contain many outliers.

3. A human being can make abrupt movements even while remaining in place; therefore, to reliably and accurately recognize abrupt movements, one should generally recognize individual parts of the body (hands, feet, head, etc.), which makes the problem of low-level analysis of video images an order of magnitude more difficult.

Taking into account these considerations, we started experiments together with the development of methods of high- and low-level processing. Today, simplified algorithms have been developed for low-level processing of video images, which, in spite of their simplicity, provide quality sufficient for the correct operation of experimental logical methods, at least on test video clips. These algorithms are implemented in the Java language in the form of a built-in class '*ImageSubtractor*' of Actor Prolog. Below, we consider the main stages of low-level processing of video images and their implementation in Actor Prolog.

The first stage of video processing is subtraction of the background. The '*ImageSubtractor*' class receives a sequence of video image frames and calculates the mathematical expectation and variance of each individual pixel of the image. Pixels the values of which differ from the mathematical expectation by a value greater than the given threshold (the threshold is set in the number of mean square deviations) are considered to be elements of foreground objects. The variance and the mathematical expectation of the values of pixels can be constantly refined during the operation of the program or on the basis of the given number of first frames. Before calculation of the mathematical expectation and variance, the images can be converted to the grayscale format, as well as processed by a two-dimensional Gaussian filter to reduce digital noise. Moreover, after the background subtraction, an image can be additionally processed by a two-dimensional rank filter. The filter counts the number of foreground pixels around every pixel on the foreground and, if this number does not exceed the set threshold, excludes the pixel from consideration.

The second stage of low-level processing of video images is the discrimination of blobs. To distinguish blobs, I developed and implemented a simplified algorithm that creates rectangular blobs. The idea of the algorithm consists in the pixels of foreground objects being outlined by rectangular frames; in turn, intersecting frames are also outlined by frames, a process that is continued until all pixels of foreground objects are outlined by disjoint rectangles. The rectangles thus calculated are considered to be the sought blobs of a video image.

The third stage of processing is the creation of tracks—the trajectories of blobs. It is assumed that successive frames of a video image contain the same blob if the rectangles of blobs in adjacent frames intersect. If there are several intersecting blobs, the blob with the greatest intersection area is taken as a continuation of a track, while all the other blobs are assumed to belong to other tracks that intersect the given track. It is assumed that the track for which the corresponding blobs are not found in the next frame still exists for a certain period of time. If a continuation of the track is not found during this period, the track will be considered complete. To

save memory, complete tracks are stored for a certain period of time and then are forgotten. In addition, tracks the length of which is less than a given threshold are considered to be artifacts and are rejected immediately. Simultaneously with the construction of tracks, their intersection points are stored. This is necessary for constructing a graph of links between blobs, as well as to accurately evaluate the velocity of individual blobs: to evaluate the velocity of blobs, one chooses regions of tracks that are free of intersections with other tracks.

The fourth stage of processing consists in determination of the velocity of blobs. The fast algorithm developed by the author calculates the lower bound of the blob velocity. For this purpose, on the basis of a given inverse matrix of the projective transformations, one calculates the physical coordinates of four corners of a rectangular blob. The numerical differentiation of coordinates gives a rough estimate for the velocity of each corner of the blob on the basis of the assumption that the point considered is at the level of the floor. It is assumed that this is so for at least one corner of the blob and that, for all the other corners of the blob, these velocity estimates turn out to be overstated since the camera is higher than the floor level and the upper part of a human visually corresponds to farther points of the plane of the floor. Based on these assumptions, one takes the least of the four calculated values of blob corner velocities as the lower bound for the velocity of the object. To increase the stability of the estimates, the values of the coordinates of blob corners, as well as intermediate and final values of the blob velocity, are subjected to median filtration. Of course, the algorithm described gives only rough estimates for the velocity of people in a frame; however, experiments with real video images have shown that the algorithm is quite suitable to distinguish between, for example, running and quiet walking.

The fifth stage of low-level processing consists in the construction of a set of connected graphs of motions of blobs. Each graph includes all tracks that had intersection points at some instants of time. If necessary, the algorithm allows one to rule out the tracks of stationary and very slowly moving objects.

The built-in class '*ImageSubtractor*' of Actor Prolog implements the above-listed stages of low-level analysis and gives the results of analysis in the form of composite terms. In the example considered of intelligent video monitoring, we use the following values of arguments of the '*ImageSubtractor*' class instance constructor:

```
('ImageSubtractor',
    extract_blobs= 'yes',
    track_blobs= 'yes',
    use_grayscale_colors= 'yes',
    apply_gaussian_filtering_to_background= 'yes',
    background_gaussian_filter_radius= 1,
    background_standard_deviation_factor= 1.2,
    apply_rank_filtering_to_background= 'yes',
    background_rank_filter_threshold= 4,
    minimal_track_duration= 5,
    inverse_transformation_matrix= [
          [0.3945,0.0468,0.0168],
          [0.0996,-0.1625,0.00056],
          [-34.0116,28.5636,1.0000]],
    sampling_rate,
    apply_median_filtering_to_velocity= 'yes',
    velocity_median_filter_halfwidth= 3);
```

The slots *extract_blobs* and *track_blobs* switch on the mode of calculation of blobs and tracks. The slot *use_grayscale_colors* switches on the conversion of images to grayscale format. The slots *apply_gaussian_filtering_to_background* and *background_gaussian_filter_radius* switch on processing of images by a Gaussian filter with a diameter of three pixels. The slot *background_standard_deviation_factor* sets a threshold for background subtraction. The slots *apply_rank_filtering_to_background* and *background_rank_filter_threshold* switch on a two-dimensional rank filter of foreground pixels and set the threshold of the filter. The *minimal_track_duration* slot sets the minimum length of a track (in frames). The *inverse_transformation_matrix* slot sets a value of the inverted matrix of projective transformations. The slot *sampling_rate* sets the frame rate of a video image. The slots *apply_median_filtering_to_velocity* and *velocity_median_filter_halfwidth* switch on median filtration of the coordinate and velocity and set the width of the window of the median filter to seven samples.

Consider the operation of '*ImagePreprocessor*,' which carries out the video image input and the entire low-level processing.

```
CLAUSES:
goal:-!,
     Time0== ?milliseconds(),
     state ? set_beginning_time(Time0),
     set_period(1/sampling_rate,0),
     activate.
```

The class '*ImagePreprocessor*' is a descendant of the built-in class '*Timer*' that implements the call of a *tick* predicate over given time intervals. The adjustment of an instance of the class is performed during its construction when the *goal* predicate is called. In this case, the current time in milliseconds (the built-in predicate *milliseconds*) is calculated and stored in a local database *state* and then the period of calls of the *tick* predicate (the built-in predicate *set_period* with arguments *1 / sampling_rate*, a period in seconds, and 0, delay before the first call of *tick*) is set, after which the instance of the '*Timer*' class is activated.

```
tick:-
     T2== ?milliseconds(),
     state ? get_beginning_time(T1),
     Delta== (T2 - T1) / 1000.0 * sampling_rate,
     N== ?convert_to_integer(?round(Delta)),!,
     load_figure(N,T2).
```

With each call of the *tick* predicate, the time (in milliseconds) that elapsed after the beginning of input of a video image is calculated (the time of the beginning of input is stored in the local data base *state* and is retrieved by the *get_beginning_time* predicate). The time in milliseconds is then recalculated to the number of a frame and is transferred to the *load_figure* predicate as an argument.

```
load_figure(N2,_):-
     state ? get_current_frame(N1),
     N1 == N2,!.
load_figure(N,_):-
     state ? set_current_frame(N),
     ShortFileName== text?format("%03d",N) + ".jpg",
     ImageToBeLoaded==
          "jar:" + data_directory + "/" +
          target_directory + "_jpg" + "/JPEGS/" +
          target_directory + ShortFileName,
     image ? does_exist(ImageToBeLoaded),!,
     image ? load(ImageToBeLoaded),
     subtractor ? subtract(N,image),
     stage_two [<<] draw_scene().
load_figure(_,T2):-
     state ? set_beginning_time(T2),
     subtractor ? reset_results.
```

The *load_figure* predicate compares the specified number of a frame with the number of the previously processed frame and, if they do not coincide, loads the frame into memory from the file with an appropriate name. The names of files are created automatically according to the rules of naming used in the CAVIAR collection of video clips [19]. Note the prefix "jar:" added at the beginning of a file name. This prefix is used because all graphic files in this example are packed in a single JAR archive with an executable Java code. A loaded image is physically stored in an instance of the class *'BufferedImage'* (the slot *image*). Then, in an instance of the class '*SynchronizedImageSubtractor*' (the slot *subtractor*), a built-in predicate *subtract* is called, the first argument of which is the number of the frame processed and the second argument is the image loaded from a file. The class '*SynchronizedImageSubtractor*' performs low-level processing of the image in the corresponding instance of the class '*ImageSubtractor*'; all the results of processing remain in the internal arrays of the instance of the class '*ImageSubtractor*' and can be retrieved as required.

Later, an asynchronous message *draw_scene()* is sent to the *stage_two* process. Note the square brackets enclosing the operator <<. The operator [<<] in Actor Prolog implements unbuffered asynchronous interaction. This means that the *stage_two* process will always

process the latest *draw_scene* message addressed to it. If the *stage_two* process does not have time to process all the *draw_scene* messages addressed to it, some of them will be merely ignored.

At the time when a video clip ends, when the sequence of files is exhausted, the *does_exist* predicate in the second rule of *load_figure* fails and, instead of the second rule, the third rule works; as a result, a new start time of video image inputting will be set, and all the results accumulated in an instance of the class '*ImageSubtractor*' will be cancelled (by the built-in predicate *reset_results*). Processing of the video clip will start anew.

The operation of the predicate *draw_scene*(), *which* implements high-level processing of video images in the '*ImageAnalyser*' process, is considered below.

3. HIGH-LEVEL ANALYSIS OF VIDEO IMAGES

There are a few possible approaches to logical description and analysis of the behavior of people by means of logic programming.

1. Of course, the Prolog language is, in itself, a declarative language the expressive capabilities of which allow one, for example, to formulate a problem of recognition of abnormal behavior of people in terms of analysis of the graphs of motion of blobs. However, in this way we inevitably face the need to introduce elements of fuzzy reasoning to the logical analysis. This is associated both with the ambiguity of the results of low-level analysis and the fuzziness of the scenarios of abnormal behavior themselves.

2. The simplest elements of fuzzy logical inference can easily be introduced into the Prolog language by means of standard arithmetic means of the language. For example, the predicate *is_a_running_person* for recognition of a running human considered in this section takes into account simultaneously two characteristics of blobs; namely, the average velocity of a blob and the length of a track. Combination of these two characteristics is performed by very simple fuzzy metrics described in terms of arithmetic functions. From the standpoint of the declarative semantics of the language, the procedure *is_a_running_person* is a standard formula of the first-order predicate logic.

3. The development of special logic languages to describe the behavior of objects is a more interesting approach from the point of view of mathematics; this includes, for example, the extension of the predicate logic by temporal, modal, spatial, fuzzy, and probabilistic expressive means followed by the implementation of these special languages by the means of the Prolog language; however, this direction of research falls outside the scope of the present article.

Let us return to our example of recognition of abnormal behavior. Let us describe the following scenario of abnormal behavior in terms of the logic language: "Two (or a few) people meet somewhere within the surveillance area of a camera. After that, the group of people splits up and at least one of the people runs away." This situation may testify to a street incident, fight, or robbery; therefore, it should be considered as a probable case of abnormal behavior (and an appropriate warning should be given to the video surveillance operator).

The class '*ImageSubtractor*' employs the following data structure to describe connected graphs of motion of blobs:

```
DOMAINS:
ConnectedGraph          = ConnectedGraphEdge*.
ConnectedGraphEdge      = {
        frame1: INTEGER,
        x1: INTEGER, y1: INTEGER,
        frame2: INTEGER,
        x2: INTEGER, y2: INTEGER,
        inputs: EdgeNumbers,
        outputs: EdgeNumbers,
        identifier: INTEGER,
        coordinates: TrackOfBlob,
        mean_velocity: REAL
        }.
EdgeNumbers             = EdgeNumber*.
EdgeNumber              = INTEGER.
TrackOfBlob             = BlobCoordinates*.
BlobCoordinates         = {
        frame: INTEGER,
        x: INTEGER, y: INTEGER,
        width: INTEGER, height: INTEGER,
        velocity: REAL
        }.
```

That is, a graph is represented as a list of underdetermined sets [10] that describe individual edges of the graph. Each edge is directed and equipped with the following attributes: the numbers of the first and the last frames (*frame1*, *frame2*), the coordinates of the first and the last point (*x1*, *y1*, *x2*, *y2*), the list of the numbers of edges that are immediate predecessors of the given edge (*inputs*), the list of the numbers of edges that are direct successor of the given edge (*outputs*), the identifier of an appropriate blob (*identifier*), the list of underdetermined sets that describe the coordinates and velocity of the blob at different instants of time (*coordinates*), and the mean velocity of the blob on the given edge of the graph (*mean_velocity*).

Let us define the predicate *is_a_kind_of_a_running_away* from the class '*ImageAnalyser*,' which describes the sought subgraph of motions of blobs:

```
is_a_kind_of_a_running_away([E2|_],G,E1,E2,E3):-
    E2 == {inputs:O,outputs:B|_},
    B == [_,_|_],
    contains_a_running_person(B,G,E3),
    is_a_meeting(O,G,E2,E1),!.
is_a_kind_of_a_running_away([_|R],G,E1,E2,E3):-
    is_a_kind_of_a_running_away(R,G,E1,E2,E3).
contains_a_running_person([N|_],G,P):-
    get_edge(N,G,E),
    is_a_running_person(E,G,P),!.
contains_a_running_person([_|R],G,P):-
    contains_a_running_person(R,G,P).
is_a_meeting(O,_,E,E):-
    O == [_,_|_],!.
is_a_meeting([N1|_],G,_,E2):-
    get_edge(N1,G,E1),
    E1 == {inputs:O|_},
    is_a_meeting(O,G,E1,E2).
get_edge(1,[Edge|_],Edge):-!.
get_edge(N,[_|Rest],Edge):-
    N > 0,
    get_edge(N-1,Rest,Edge).
```

In other words, the graph describes a running away situation if it contains an edge *E2*, which has edge *E3* corresponding to a person who is running away as a successor and edge *E1* corresponding to the meeting of at least two people as a predecessor. Note that the edge *E2* should have at least two direct successors (test *B* == [_, _|_]).

```
is_a_running_person(E,_,E):-
    E == {mean_velocity:V,frame1:T1,frame2:T2|_},
    M1== ?fuzzy_metrics(V,1.0,0.5),
    D== (T2 - T1) / sampling_rate,
    M2== ?fuzzy_metrics(D,0.75,0.5),
    M1 * M2 >= 0.5,!.
is_a_running_person(E,G,P):-
    E == {outputs:B|_},
    contains_a_running_person(B,G,P).
```

An edge of the graph corresponds to a running person if the mean velocity and the length of the track of a blob correspond to the given fuzzy definition. An auxiliary function for calculating the fuzzy metrics is given below (the first argument is the value to be estimated, the second is a given threshold, and the third is the width of the uncertainty area).

```
fuzzy_metrics(X,T,H) = 1.0 :-
     X >= T + H,!.
fuzzy_metrics(X,T,H) = 0.0 :-
     X <= T - H,!.
fuzzy_metrics(X,T,H) = V :-
     V== (X-T+H) * (1 / (2*H)).
```

Fig. 2. The intelligent video surveillance program has recognized a situation like a street incident. Heavy solid lines show the trajectories of people involved in a probable street conflict, thin lines and circles indicate connected graphs of motions of blobs, and rectangles represent the people involved in the incident.

Using the *is_a_kind_of_a_running_away* predicate, we implement the above-mentioned procedure *draw_scene()* of the '*ImageAnalyser*' class, which performs a logical analysis of the scene and the visualization of the results of intelligent video surveillance:

```
draw_scene():-
    subtractor ? commit,
    subtractor ? get_recent_frame_number(N),
    subtractor ? get_recent_image(image),
    subtractor ? get_connected_graphs(Graphs),
    graphic_window ? suspend_redrawing,
    graphic_window ? clear,
    graphic_window ? draw_image(image,0,0,1,1),
    image ? get_size_in_pixels(IW,IH),
    draw_graphs(IW,IH,Graphs,N),
    graphic_window ? draw_now.
```

The built-in predicate *commit* of the '*ImageSubtractor*' class calculates the graphs of motion of blobs in the state at the moment of the last frame processed by the class. This operation is singled out into a separate predicate with a view to optimization; the program may have time to process several frames of the video image before it will need the graphs of motion of blobs. The built-in predicate *get_recent_frame_number* returns the number of the last frame processed by the moment of calling the predicate *commit*. The built-in predicate *get_recent_image* returns the corresponding image (the data are recorded in an instance of the '*BufferedImage*' class that is stored in the *image* slot). The built-in predicate *get_connected_graphs* returns the list of connected graphs of motions of blobs. We use the following predicates of the built-in class '*Canvas2D*': *suspend_redrawing*—suspend the redrawing of an image in the window, *clear*—clear the window, *draw_image*—draw an image (the first argument is an instance of the class '*BufferedImage*,' the second and third are the coordinates of the upper left corner of the image, and the fourth and the fifth are the width and the height of the image in a scale from 0 to 1), and *draw_now*—resume the redrawing of the image in the window. The predicate *draw_graphs* analyzes the graphs of motions of blobs and displays additional information on the screen in the form of lines, rectangles, and inscriptions. Its arguments are the width and height of the image (obtained by the *get_size_in_pixels* predicate of the built-in class '*BufferedImage*'), the list of the graphs, and the frame number.

```
draw_graphs(IW,IH,[G|R],N):-
    is_a_kind_of_a_running_away(G,G,_,_,_),!,
    draw_graph(IW,IH,G,G,'yes',N),
    graphic_window ? set_text_alignment('CENTER','CENTER'),
```

```
    graphic_window ? set_font({size:64}),
    graphic_window ? set_pen({color:'Yellow'}),
    graphic_window ? draw_text(0.5,0.5,"Attention!"),
    draw_graphs(IW,IH,R,N).
draw_graphs(IW,IH,[G|R],N):-!,
    draw_graph(IW,IH,G,G,'no',N),
    draw_graphs(IW,IH,R,N).
draw_graphs(_,_,_,_).
```

The predicate *draw_graphs* unrolls the list of graphs, analyzes them, draws the trajectories of blobs on the screen using the predicate *draw_graph*, and outputs the results of logical analysis. If a current graph contains the sought scenario of a street incident, the execution of the predicate *is_a_kind_of_a_running_away* ends successfully, and the first rule of the predicate *draw_graphs* works on. A word "Attention!" appears on the screen, and the predicate *draw_graph* indicates the current position of the participants of the incident (Fig. 2). For simplicity, here we do not consider the details of the operation of graphic predicates of Actor Prolog, as well as the internal structure of the *draw_graph* predicate. The program of intelligent video surveillance yields the trajectories of people involved in a probable street conflict (they are indicated by heavy lines in the figure), as well as their current positions (people are shown by rectangles).

The example considered represents a simple logic program that performs intelligent video surveillance of abnormal behavior of people. The program implements all the necessary stages of information processing: real-time inputting of video information, low-level processing of a video image, logical analysis of video information, and displaying the results on the screen.

CONCLUSIONS

We have developed a method for intelligent video monitoring of abnormal behavior of people on the basis of parallel object-oriented logic programming. We have shown that the logic language Actor Prolog allows one to describe and recognize probable scenarios of abnormal behavior of people by providing a natural separation and implementation of various stages of video image processing in the form of interacting parallel processes. Translation of the logic programs of intelligent video monitoring into the Java language provides necessary reliability, portability, and openness of the software, including the possibility of access to open-code libraries.

We have designed and implemented in the Java language an experimental built-in class of Actor Prolog that carries out low-level processing of video images, which is necessary for their subsequent logical analysis. This class is built into an open source library [20], which is designed with a view to facilitate experiments with logical analysis of video images for third-party designers and cooperation in this promising field of research.

ACKNOWLEDGMENTS

I am grateful to Abhishek Vaish (IIIT Allahabad) and to A.F. Polupanov, O.S. Sushkova, and V.E. Antciperov from the Kotel'nikov Institute of Radio Engineering and Electronics, Russian Academy of Sciences, as well as to V.V. Devyatkov, A.N. Alfimtsev, V.S. Popov, and I.I. Lychkov from Bauman Moscow State Technical University.

This work was supported by the Russian Foundation for Basic Research, project no. 13-07-92694-Ind_a.

REFERENCES

1. J. Aggarwal and M. Ryoo, "Human activity analysis: A review," ACM Comput. Surv. (CSUR) **43** (3), 16:1–16:43 (2011).
2. J. Junior, S. Musse, and C. Jung, "Crowd analysis using computer vision techniques: A survey," IEEE Signal Processing Mag. **27** (5), 66–77 (2010).
3. I. Kim, H. Choi, K. Yi, et al., "Intelligent visual surveillance: A survey," Int. J. Control, Automat., Syst. **8** (5), 926–939 (2010).
4. A. Artikis, A. Skarlatidis, and G. Paliouras, "Behaviour recognition from video content: a logic programming approach," Int. J. Artificial Intellig. Tools **19** (2), 193–209 (2010).
5. A. Skarlatidis, A. Artikis, J. Filippou, and G. Paliouras, "A probabilistic logic programming event calculus," Theory Practice Logic Programming, No. 9, 1–33 (2014).
6. V. Shet, M. Singh, C. Bahlmann, et al., "Predicate logic based image grammars for complex pattern recognition," Int. J. Comput. Vision **93** (2), 141–161 (2011).
7. V. Shet, D. Harwood, and L. Davis, "VidMAP: Video monitoring of activity with Prolog," in *Proc. IEEE Conf. AVSS 2005* (Como, 2005), pp. 224–229.
8. S. O'Hara, "VERSA: Video event recognition for surveillance applications," *M.S. Thesis* (Univ. of Nebraska at Omaha, 2008).
9. F. Machot, K. Kyamakya, B. Dieber, and B. Rinner, "Real time complex event detection for resource-limited multimedia sensor networks," in *Proc. AMMCSS 2011* (Klagenfurt, 2011), pp. 468–473.

10. A. A. Morozov, "Actor Prolog: An object-oriented language with the classical declarative semantics," in *Proc. IDL 1999*, Ed. by K. Sagonas and P. Tarau (Paris, 1999), pp. 39–53. http://www.cplire.ru/Lab144/paris.pdf
11. A. A. Morozov, "On semantic link between logic, object-oriented, functional, and constraint programming," in *Proc. MultiCPL 2002* (Ithaca, NY, 2002), pp. 43–57. http://www.cplire.ru/Lab144/multicpl.pdf
12. A. A. Morozov, "Logic object-oriented model of asynchronous concurrent computations," Pattern Recogn. Image Anal. **13** (4), 640–649 (2003). http://www.cplire.ru/Lab144/pria640.pdf
13. A. A. Morozov, "Development and application of logical actors mathematical apparatus for logic programming of web agents," in *Proc. ICLP 2003*, Ed. by C. Palamidessi (Springer, Heidelberg, 2003), pp. 494–495.
14. A. A. Morozov, "Operational approach to the modified reasoning, based on the concept of repeated proving and logical actors," in *Proc. CICLOPS 2007*, Ed. by V. S. C. Salvador Abreu (Porto, 2007), pp. 1–15. http://www.cplire.ru/Lab144/ciclops07.pdf
15. A. A. Morozov, "Visual logic programming method based on structural analysis and design technique," in *Proc. ICLP 2007*, Ed. by V. Dahl and I. Niemel (Springer, Heidelberg, 2007), pp. 436–437.
16. A. A. Morozov and A. F. Polupanov, "Intelligent visual surveillance logic programming: implementation issues," in *Proc. CICLOPS-WLPE 2014*, Ed. by T. Ströder and T. Swift, *Aachener Informatik Berichte no. AIB-2014-09* (RWTH Aachen Univ., 2014), pp. 31–45. http://aib.informatik.rwth-aachen.de/2014/2014-09.pdf
17. A. A. Morozov, A. Vaish, A. F. Polupanov, et al., "Development of concurrent object-oriented logic programming system to intelligent monitoring of anomalous human activities," in *Proc. BIODEVICES 2014*, Ed. by A. C. G. Plantier, Jr., T. Schultz, et al. (SCITEPRESS, 2014), pp. 53–62.
18. A. A. Morozov and O. S. Sushkova, The intelligent visual surveillance logic programming Web Site (2014). http://www.fullvision.ru/actor_prolog_2014
19. R. Fisher, CAVIAR test case scenarios. The EC funded project IST 2001 37540 (2007). http://homepages.inf.ed.ac.uk/rbf/CAVIAR/
20. A. A. Morozov, A GitHub repository containing source codes of Actor Prolog built-in classes (2014). https://github.com/Morozov2012/actor-prolog-java-library

Translated by I. Nikitin

Aleksei A. Morozov was born in 1968. Graduated from Bauman Moscow State Technical University in 1991. Received his Candidate's Degree in Physics and Mathematics in 1998. Senior researcher at the Kotel'nikov Institute of Radio Engineering and Electronics, Russian Academy of Sciences. His scientific interests are logic programming, intelligent video surveillance, and processing of biomedical signals (EEG, EMG, and MEG). The author of 90 publications.