

Intelligent Visual Surveillance Logic Programming: Implementation Issues

Alexei A. Morozov^{1,2} and Alexander F. Polupanov^{1,2}

¹ Kotel'nikov Institute of Radio Engineering and Electronics of RAS,
Mokhovaya 11, Moscow, 125009, Russia

² Moscow State University of Psychology & Education,
Sretenka 29, Moscow, 107045, Russia
`morozov@cplire.ru, sashap55@mail.ru`

Abstract. The main idea of the logic programming approach to the intelligent video surveillance is in using a first order logic for describing complex events and abstract concepts like anomalous human activity, i.e. brawl, sudden attack, armed attack, leaving object, loitering, pick pocketing, personal theft, immobile person, etc. We consider main implementation issues of our approach to the intelligent video surveillance logic programming: object-oriented logic programming of concurrent stages of video processing, translating video surveillance logic programs to fast Java code, embedding low-level means for video storage and processing to the logic programming system.

Keywords: intelligent visual surveillance, object-oriented logic programming, concurrent logic programming, abnormal behavior detection, anomalous human activity, Actor Prolog, complex events recognition, computer vision, technical vision, Prolog to Java translation

1 Introduction

Human activity recognition is a rapidly growing research area with important application domains including security and anti-terrorist issues [1,11,12]. Recently logic programming was recognized as a promising approach for dynamic visual scenes analysis [7,24,23,14,22]. The idea of the logic programming approach is in usage of logical rules for description and analysis of people activities. Knowledge about object co-ordinates and properties, scene geometry, and human body constraints is encoded in the form of certain rules in a logic programming language and is applied to the output of low-level object / feature detectors. There are several studies based on this idea. In [7] a system was designed for recognition of so-called long-term activities (such as fighting and meeting) as temporal combinations of short-term activities (walking, running, inactive, etc.) using a logic programming implementation of the Event Calculus. The ProbLog probabilistic logic programming language was used to handle the uncertainty that occurs in human activity recognition. In [24] an extension of predicate logic with

the bilattice formalism that permits processing of uncertainty in the reasoning was proposed. The VidMAP visual surveillance system that combines real time computer vision algorithms with the Prolog based logic programming had been announced by the same team. In [23] the VERSA general-purpose framework for defining and recognizing events in live or recorded surveillance video streams is described. According to [23], VERSA ensures more advanced spatial and temporal reasoning than VidMAP and is based on SWI-Prolog. In [14] a real time complex audio-video event detection based on Answer Set Programming approach is proposed. The results indicate that this solution is robust and can easily be run on a chip.

The distinctive feature of our approach to the visual surveillance logic programming is in application of general-purpose concurrent object-oriented logic programming features to it, but not in the development of a new logical formalism. We use the Actor Prolog object-oriented logic language [15,16,17,19,18] for implementation of concurrent stages of video processing. A state-of-the-art Prolog-to-Java translator is used for efficient implementation of logical inference on video scenes. Special built-in classes of the Actor Prolog language were developed and implemented for the low-level video storage and processing.

Basic principles of video surveillance logic programming are described in Section 2. The concurrent logic programming issues linked with the video processing are described in Section 3. The problems of efficient implementation of logical inference on video scenes and translation of logic programs to fast Java code are discussed in Section 4. The problems of video storage and low-level processing are considered in Section 5.

2 Basic principles of video surveillance logic programming

A common approach to human activity recognition includes low-level and high-level stages of the video processing. An implementation of the logic programming approach requires consideration of different mathematical and engineering problems on each level of recognition:

1. Low-level processing stages usually include recognition of moving objects, pedestrians, faces, heads, guns, etc. The output of the low-level procedures is an input for the high-level (semantic) analyzing procedures. So, the low-level procedures should be fast enough to be used for real time processing and the high-level logical means should utilize effectively and completely the output of the low-level procedures.
2. Adequate high-level logical means should be created / used to deal with temporal and spatial relationships in the video scene, as well as uncertainty in the results of low-level recognition procedures.
3. A proper hierarchy of the low-level and high-level video processing procedures should be constructed. The procedures of different levels are usually implemented on basis of different programming languages.
4. Selection of logic programming system for implementation of the logical inference is of great importance, because it should provide high-level syntax

means for implementation of required logical expressions as well as computation speed and efficiency in real time processing of big amount of data.

Let us consider an example of logical inference on video. The input of a logic program written in Actor Prolog is the *Fight_OneManDown* standard sample provided by the CAVIAR team [8]. The program will use no additional information about the content of the video scene, but only co-ordinates of 4 defining points in the ground plane (the points are provided by CAVIAR), that are necessary for estimation of physical distances in the scene.



Fig. 1. An example of CAVIAR video with a case of a street offence: one person attacks another.

The video (see Fig. 1) demonstrates a case of a street offence—a probable conflict between two persons. These people meet in the scope of the video camera, then one person attacks another one, the second person falls, and the first one runs away. This incident could be easily recognized by a human; however an attempt to recognize it automatically brings to light a set of interesting problems in the area of pattern recognition and video analysis.

First of all, note that probably the main evidence of an anomalous human activity in this video is so-called abrupt motion of the persons. Abrupt motions can be easily recognized by a human as motions of a person's body and / or arms and legs with abnormally high speed / acceleration. So, a logic programmer has a temptation to describe an anomalous human activity in terms of abrupt motions, somehow like this: "Several persons have met sometime and somewhere. After that they perform abrupt motions. This is probably a case of a street fighting." It is not a problem to implement this definition in Prolog using a set of logical rules, however real experiments with video samples show that this naive approach is an impractical one or simply does not work. The problem is that in the general case computer low-level procedures recognize abrupt motions much worse than a human and there are several serious reasons for this:

1. Generally speaking, it is very difficult to determine even the exact co-ordinates of a person in a video scene. A common approach to the problem is

usage of so-called ground plane assumption, that is, the computer determines co-ordinates of body parts that are situated inside a pre-defined plane and this pre-defined plane usually is a ground one. So, one can estimate properly the co-ordinates of person's shoes, but a complex surface of a ground and / or presence of stairs and other objects, etc. make the problem much more complex.

2. Even computing the first derivative of moving person's co-ordinates is a problem usually, because the silhouette of the person changes unpredictably in different lighting conditions and can be partially overlapped by other objects. As a consequence, the trajectory of a person contains a big amount of false co-ordinates that makes numerical differentiation senseless.
3. One can make abrupt motions even standing still. This means that in the general case the program should recognize separate parts of person's body to determine abrupt motions in a robust and accurate way.

All these issues relate to the low-level video processing and probably are not to be implemented in a logic language. Nevertheless, they illustrate a close connection between the principles to be used for logical description of anomalous human activity and the output of low-level video processing procedures. We take into account this connection in our research, when we address the problem of the high-level semantic analysis of people activities.

In the example under consideration, we will solve the problem of anomalous human activity recognition using automatic low-level algorithms that trace persons in video scene and estimate average velocity in different segments of the trajectories [22]. This low-level processing includes extraction of foreground blobs, tracking of the blobs over time, detection of interactions between the blobs, creation of connected graphs of linked tracks of blobs, and estimation of average velocity of blobs in separate segments of tracks. This information is received by the logic program in a form of Prolog terms describing the list of connected graphs. We will use the following data structures for describing connected graphs of tracks³:

```
DOMAINS:
ConnectedGraph = ConnectedGraphEdge*.
ConnectedGraphEdge = {
    frame1: INTEGER,
    x1: INTEGER, y1: INTEGER,
    frame2: INTEGER,
    x2: INTEGER, y2: INTEGER,
    inputs: EdgeNumbers,
    outputs: EdgeNumbers,
    identifier: INTEGER,
    coordinates: TrackOfBlob,
```

³ Note, that the DOMAINS, the PREDICATES, and the CLAUSES program sections in Actor Prolog have traditional semantics developed in the Turbo / PDC Prolog system.

```

        mean_velocity: REAL
    }.
EdgeNumbers = EdgeNumber*.
EdgeNumber = INTEGER.
TrackOfBlob = BlobCoordinates*.
BlobCoordinates = {
    frame: INTEGER,
    x: INTEGER, y: INTEGER,
    width: INTEGER, height: INTEGER,
    velocity: REAL
}.

```

That is, connected graph of tracks is a list of underdetermined sets [15] denoting separate edges of the graph. The nodes of the graph correspond to points where tracks cross, and the edges are pieces of tracks between such points. Every edge is directed and has the following attributes: numbers of first and last frames (*frame1*, *frame2*), co-ordinates of first and last points (*x1*, *y1*, *x2*, *y2*), a list of edge numbers that are direct predecessors of the edge (*inputs*), a list of edge numbers that are direct followers of the edge (*outputs*), the identifier of corresponding blob (an integer *identifier*), a list of sets describing the co-ordinates and the velocity of the blob in different moments of time (*coordinates*), and an average velocity of the blob in this edge of the graph (*mean_velocity*).

The logic program will check the graph of tracks and look for the following pattern of interaction among several persons: “If two or more persons met somewhere in the scene and one of them has run after the end of the meeting, the program should consider this scenario as a kind of a running away and a probable case of a sudden attack or a theft.” So, the program will alarm if this kind of sub-graph is detected in the total connected graph of tracks. In this case, the program marks all persons in the inspected graph by yellow rectangles and outputs the “Attention!” warning in the middle of the screen (see Fig. 2).

One can describe the concept of a running away formally using defined connected graph data type:

```

PREDICATES:
is_a_kind_of_a_running_away(
    ConnectedGraph,
    ConnectedGraph,
    ConnectedGraphEdge,
    ConnectedGraphEdge,
    ConnectedGraphEdge) - (i,i,o,o,o);

```

We will define the *is_a_kind_of_a_running_away*(*G*, *G*, *P1*, *E*, *P2*) predicate with the following arguments: *G*—a graph to be analyzed (note that the same data structure is used in the first and the second arguments), *E*—an edge of the graph corresponding to a probable incident, *P1*—an edge of the graph that is a predecessor of *E*, *P2*—an edge that is a follower of *E*. Note that *G* is an input



Fig. 2. A logical inference has found a possible case of a sudden attack in the graph of blob trajectories. All probable participants of the conflict are marked by yellow rectangles. The tracks are depicted by lines.

argument of the predicate and $P1$, E , and $P2$ are output ones. Here is an Actor Prolog program code with brief explanations:

```
CLAUSES:
is_a_kind_of_a_running_away([E2|_],G,E1,E2,E3):-
    E2 == {inputs:0,outputs:B|_},
    B == [_|_|],
    contains_a_running_person(B,G,E3),
    is_a_meeting(0,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(0,_,E,E):-
    0 == [_|_|],!.
is_a_meeting([N1|_],G,_,E2):-
    get_edge(N1,G,E1),
    E1 == {inputs:0|_},
    is_a_meeting(0,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 contains a case of a running away if there is an edge $E2$ in the graph that has a follower $E3$ corresponding to a running person and predecessor $E1$ that corresponds to a meeting of two or more persons. It is

requested also that $E2$ has two or more direct followers (it is a case of branching in the graph). Note, that in the Actor Prolog language, the `==` operator corresponds to the `=` ordinary equality of the standard Prolog.

A fuzzy definition of the running person concept is as follows:

```
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).
```

The graph edge corresponds to a running person if the average velocity and the length of the track segment correspond to the fuzzy definition. Note that Actor Prolog implements a non-standard functional notation, namely, the `?` prefix informs the compiler that the *fuzzy_metrics* term is a call of a function, but not a data structure.

An auxiliary function that calculates the value of the fuzzy metrics is represented below. The first argument of the function is a value to be checked, the second argument is a value of a fuzzy threshold, and the third one is the width of the threshold ambiguity area. The `=` delimiter defines an extra output argument that is a result to be returned by the function:

```
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)).
```

Note that 37 lines of the Actor Prolog code considered above correspond to 301 lines of the optimized Java source code implementing graph search operations and this is perhaps the best demonstration of the reasons why the Prolog language is a good choice for this application.

This example illustrates the basic principles of logical description of anomalous human activity and logical inference on video data. However, even a simplest scheme of video surveillance logic program should contain much more elements, including video information gathering, low-level image analysis, high-level logical inference control, and reporting the results of intelligent visual surveillance.

3 Concurrent video processing

It is a good idea to divide a visual surveillance logic program to concurrent sub-processes implementing different stages of video processing, because the working

intensity of different sub-processes is various. For instance, video data gathering and low-level analysis require a big amount of computational resources and other sub-processes that implement high-level analysis and visualizing results of video surveillance could wait for the output of the former sub-process.

In the example under consideration, we will create two concurrent processes with different priorities⁴. The first process has higher priority and implements video data gathering. This process reads JPEG files and sends them to the instance of the *'ImageSubtractor'* predefined class that implements all low-level processing of video frames. The sampling rate of the video is 25 frames per second, so the process loads a new JPEG file every 40 milliseconds. The second concurrent process implements logical analysis of collected information and outputs results of the analysis. The analysis of video frames requires more computational resources, but it does not suspend the low-level analysis, because the second process has less priority. The analysis includes creation of connected graphs of linked tracks of blobs and estimation of average velocity of blobs in separate segments of tracks. This information is received by the logic program in a form of connected graphs described in the previous section.

The total text of the logic program is not given here for brevity; we will discuss only the structure of the main class of the program.

```
class 'Main' (specialized 'Alpha'):
constant:
    data_directory    = "data";
    target_directory  = "Fight_OneManDown";
    sampling_rate     = 25.0;
    stage_one = (('ImagePreprocessor',
                  data_directory,
                  target_directory,
                  sampling_rate,
                  low_level_analyzer,
                  stage_two));
    stage_two = (('ImageAnalyzer',
                  low_level_analyzer,
                  sampling_rate));
internal:
    low_level_analyzer = ('ImageSubtractor',
                          extract_blobs= 'yes',
                          track_blobs= 'yes',
                          ...);
```

The *'Main'* class has the *stage_one* and the *stage_two* slots⁵ containing two above mentioned processes. The *low_level_analyzer* slot contains an instance of the *'ImageSubtractor'* built-in class implementing low-level video analysis procedures (see Fig. 3). The *data_directory*, the *target_directory*, and the

⁴ See [17] for details of Actor Prolog model of asynchronous concurrent computations.

⁵ The slot is a synonym for the instance variable in the Actor Prolog language.

sampling_rate auxiliary slots contain information about source data files location and the sampling rate of the video clip.

The idea of the schema is the following: the instance of the '*ImageSubtractor*' built-in class is used as a container for storing video data and intermediate results of the low-level processing. The instances of the '*ImagePreprocessor*' and the '*ImageAnalyzer*' classes accept the *low_level_analyzer* slot as an argument and use it for the inter-process data exchange. So, the object-oriented features give us an essential solution of the problem of big data storage in the logic language; all low-level video data is encapsulated in the special class and the logic program handles medium-size terms describing the results of the low-level analysis only.

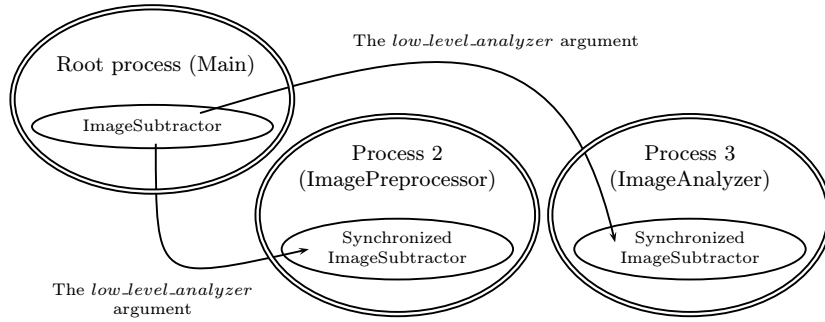


Fig. 3. The '*SynchronizedImageSubtractor*' class implements synchronization of three concurrent processes and ensures a safe access to the internal data arrays of the '*ImageSubtractor*' class instance.

The '*ImagePreprocessor*' class implements video data gathering and all low-level processing of video frames. The '*ImageAnalyzer*' class contains the rules considered in the previous section, implements high-level stage of video analysis, and outputs the results of intelligent video surveillance.

Note, that Actor Prolog prohibits a process from invoking a predicate from a class instance belonging to another process. This means that the sharing of the '*ImageSubtractor*' class instance mentioned above requires an additional care. In detail, the '*ImageAnalyzer*' class uses the '*SynchronizedImageSubtractor*' built-in class as an envelope for the '*ImageSubtractor*' class instance.

```
class 'ImageAnalyzer' (specialized 'Alpha'):
constant:
    sampling_rate;
    low_level_analyzer;
internal:
    subtractor = ('SynchronizedImageSubtractor',
                  image_subtractor= low_level_analyzer);
...
```

The '*SynchronizedImageSubtractor*' built-in class implements the same methods of low-level video processing as the '*ImageSubtractor*' class. The only difference is that the instance of the '*SynchronizedImageSubtractor*' class is created inside the '*ImageAnalyzer*' process and there are no restrictions on the usage of its methods inside this process (see Fig. 3).

Thus, the '*SynchronizedImageSubtractor*' class ensures a safe access to the internal data arrays of the '*ImageSubtractor*' class from concurrent processes and implements all necessary operations on video data. The data access synchronization is implemented inside the built-in classes, but not at the level of the programming language; Actor Prolog is asynchronous concurrent language, it has no syntactical means for concurrent processes synchronization and supports syntactically only the asynchronous inter-process communications.

4 Prolog to Java translation

A high CPU / memory consumption is a characteristic of the video processing. So, the visual surveillance application is a good way to test whether the logic programming system is mature enough to be used in the industry. One could enumerate the following requirements for a logic programming system / language selected as a basis for the visual surveillance application:

1. Firstly, the logic programming system should generate a fast executable code. A deep code optimization is absolutely necessary even for the logic programs that use 2D graphic intensively for reporting intelligent visual surveillance results.
2. The executable code should be robust; absence of any memory leak should be guaranteed. An incorrect memory handling and / or incorrect concurrent access to data structures produce an unavoidable crash of the program.
3. The logic programming system should be an open one; the extension of the system by specialized classes / procedures implementing low-level processing should be easy and transparent for application programmers.

These requirements are contradictory because considerable code optimization implies usage of complex compilation algorithms and low-level code generation that are potential reasons for difficult-to-locate errors, memory leaks, and unstable operation of executable code. Even if the compiler is well-debugged the permanent development of built-in classes and libraries is a constant potential source of such errors. There is a fundamental contradiction between the openness and the optimization of the programming system.

In particular, application of a compilation schema based on C / C++ intermediate code generation (Mercury [10], KLIC [9], wamcc [3]) was recognized as an appropriate way to obtain maximal speed of the executable code. On the other hand, generation of Java intermediate code (Actor Prolog [20], PrologCafe [2], KLIJava [13], SAE-Prolog [6], jProlog [5]) ensures platform independence of the application software and guarantees absence of difficult-to-locate errors caused by memory leaks and out-of-range array operations. We use a compilation from

the Actor Prolog language to Java, because, from our point of view, modern processors are fast enough to give up the speed of the executable code for the sake of robustness, readability, and openness of the logic program. Moreover, using an industrial Java virtual machine as a basis for the logic programming system ensures its flexibility and quick adaptation to new operational systems and processor architectures.

In contrast to conventional approaches, we use neither WAM (PrologCafe, wamcc) nor binarization of the logic program (jProlog, BinProlog [25]). The Actor Prolog compiler generates a kind of an idiomatic (i.e., well-readable) source code (SAE-Prolog, P# [4]), but in contrast to the SAE-Prolog project [6] we use domains / predicates declarations to process non-deterministic, deterministic, and imperative predicates in different ways. In contrast to the P# project [4] we implement non-idiomatic predicate calls from idiomatic predicates and vice versa.

The Actor Prolog logic language differs from the Clocksin&Mellish Prolog a lot. Turbo-Prolog style Domain and Predicate declarations of Actor Prolog are very important for the industrial application programming and help in executable code optimization. On the other hand, object-oriented features and supporting concurrent programming make translation of an Actor Prolog code to be a non-trivial problem.

The state-of-the-art compilation schema of the Actor Prolog system includes the following steps:

1. *Source text scanning and parsing.* Methods of thinking translation preventing unnecessary processing of already translated source files are implemented. That is, after the update of source codes, the compiler tries to use information collected / computed during its previous run. This feature is very important for the industrial programming.
2. *Inter-class links analysis.* On this stage of global analysis, the translator collects information about usage of separate classes in the program, including data types of arguments of all class instance constructors. This information is necessary for the global flow analysis and the global optimization of the program (unused predicates are eliminated from the program).
3. *Type check.* The translator checks data types of all predicate arguments and arguments of all class instance constructors.
4. *Determinism check.* The translator checks whether predicates are deterministic or non-deterministic. A special kind of so-called imperative predicates is supported, that is, the compiler can check whether a predicate is deterministic and never fails.
5. *A global flow analysis.* The compiler tracks flow patterns of all predicates in all classes of the program.
6. *Generation of an intermediate Java code.*
7. *Translation of this Java code by a standard Java compiler.*

The determinism check ensures a possibility to use different optimization methods for different kinds of predicates:

1. The imperative predicates check is the most complex stage in the translation schema, because it requires a check of all separate clauses as well as mutual influence of the clauses / predicates. Nevertheless, this check is of critical importance, because the imperative predicates usually constitute the main part of the program and the check gives information for very high level optimization of these predicates—the logic language clauses are translated to Java procedures directly.
2. Deterministic predicates are translated to Java procedures too (all clauses of one predicate correspond to one Java procedure). Backtracking is implemented using a special kind of light-weight Java exceptions.
3. Non-deterministic predicates are implemented using a standard method of continuation passing. Clauses of one predicate correspond to one or several automatically generated Java classes.

Tail recursion optimization is implemented for recursive predicates; that is critically important for the video processing applications. Recursive predicates are implemented using the *while* Java command. Moreover, the Actor Prolog language supports explicit definition of ground / non-ground domains and the translator uses this information for deep optimization of ground term unification.

Table 1. Prolog benchmark testing (Intel Core i5-2410M, 2.30 GHz, Win7, 64-bit)

Test	Iter. No.	Actor Prolog to Java	SWI-Prolog v. 7.1.10
NREV	3,000,000	109,090,909 lips	15,873,523 lips
CRYPT	100,000	1.758510 ms	2.03347 ms
DERIV	10,000,000	0.055747 ms	0.0104318 ms
POLY_10	10,000	3.756900 ms	4.3681 ms
PRIMES	100,000	0.042540 ms	0.13478 ms
QSORT	1,000,000	0.042924 ms	0.059561 ms
QUEENS(9)	10,000	17.495200 ms	31.729 ms
QUERY	10,000	3.141500 ms	0.3713 ms
TAK	10,000	4.010800 ms	10.2836 ms

The described compilation schema ensures an acceptable performance of the executable code (see Table. 1). Deterministic and imperative predicates with ground arguments are optimized quite well (the NREV test demonstrates more than 100 millions lips). At the same time, non-deterministic predicates work slowly (CRYPT, QUEENS, QUERY); this is a fundamental disadvantage of the approach based on continuation passing and translation to the high-level intermediate language, because it cannot handle possible run-time optimization of Prolog stacks. Arithmetical predicates work fast enough in Actor Prolog (PRIMES, QSORT, TAK), but there is a possibility for better optimization of symbolic computations (DERIV, POLY_10).

Note that development of standard benchmark set relevant to the visual surveillance application domain is still a challenge, because different stages of

video processing (low-level and high-level) demand different performance requirements. At present, we can demonstrate only that the Actor Prolog system is fast enough for real-time analyzing clips of the standard data set [8].

The translator creates Java classes corresponding to the classes of an object-oriented Actor Prolog program. Given external Java classes can be declared as ancestors of these automatically created classes and this is the basic principle of the implementation of built-in classes [21] and integration of Actor Prolog programs with external libraries. The possibility of easy extension of the Actor Prolog programming system by new built-in classes is a benefit of the selected implementation strategy. For instance, the Java2D and the Java3D libraries are connected with the Actor Prolog system in this way.

5 Low-level video processing

A typical intelligent video surveillance system includes high-level procedures (that is, anomalous human activity recognition) and low-level video processing procedures (for instance, background subtraction, discrimination of foreground blobs, tracking blobs over time, detection of interactions between the blobs, etc.). We have developed and implemented in Java the '*ImageSubtractor*' built-in Actor Prolog class supporting all necessary low-level procedures. This class implements the following means:

1. Video frames pre-processing including 2D-gaussian filtering, 2D-rank filtering, and background subtraction.
2. Recognition of moving blobs and creation of Prolog data structures describing the co-ordinates of the blobs in each moment.
3. Recognition of tracks of blob motions and creation of Prolog data structures describing the co-ordinates and the velocity of the blobs. The tracks are divided into separate segments; there are points of interaction between the blobs at the ends of a segment.
4. Recognition and ejection of immovable and slowly moving objects. This feature is based on a simple fuzzy inference on the attributes of the tracks (the co-ordinates of the tracks and the average velocities of the blobs are considered).
5. Recognition of connected graphs of linked tracks of blob motions and creation of Prolog data structures describing co-ordinates and velocities of blobs. We consider two tracks as linked if there are interactions between the blobs of these tracks. In some applications, it is useful to eject tracks of immovable and slowly moving objects from the graphs before further processing of the video scenes.

We have started our experiments with low-level procedures implemented in pure Java; however, it is clear that further development of video surveillance methods requires usage of advanced computer vision libraries. A promising approach for implementation of the low-level recognition procedures in a logic language is usage of the OpenCV computer vision library and we are planning to link Actor Prolog with the JavaCV library that is a Java interface to OpenCV.

6 Conclusion

We have created a research software platform based on the Actor Prolog concurrent object-oriented logic language and a state-of-the-art Prolog-to-Java translator for experimenting with the intelligent visual surveillance. The platform includes the Actor Prolog logic programming system and an open source Java library of Actor Prolog built-in classes [21]. It is supposed to be complete for facilitation of research in the field of intelligent monitoring of anomalous people activity and studying logical description and analysis of people behavior.

Our study has demonstrated that translation from a concurrent object-oriented logic language to Java is a promising approach for application of the logic programming to the problem of intelligent monitoring of people activity; the Actor Prolog logic programming system is suitable for this purpose and ensures essential separation of the recognition process into concurrent sub-processes implementing different stages of high-level analysis.

7 Acknowledgements

Authors are grateful to Abhishek Vaish, Vyacheslav E. Antciperov, Vladimir V. Deviatkov, Aleksandr N. Alfimtsev, Vladislav S. Popov, and Igor I. Lychkov for cooperation.

The valuable comments of the anonymous referees are gratefully appreciated.

We acknowledge a partial financial support from the Russian Foundation for Basic Research, grant No 13-07-92694.

References

1. Aggarwal, J., Ryoo, M.: Human activity analysis: A review. *ACM Computing Surveys (CSUR)* 43(3), 16:1–16:43 (Apr 2011), <http://doi.acm.org/10.1145/1922649.1922653>
2. Banbara, M., Tamura, N., Inoue, K.: Prolog Cafe: A Prolog to Java translator system. In: Umeda, M., Wolf, A., Bartenstein, O., Geske, U., Seipel, D., Takata, O. (eds.) *Declarative Programming for Knowledge Management*. pp. 1–11. LNAI 4369, Springer, Heidelberg (2006)
3. Codognet, P., Diaz, D.: wamcc: Compiling Prolog to C. In: Sterling, L. (ed.) *ICLP* 1995. pp. 317–331. MIT Press (1995)
4. Cook, J.J.: Optimizing P#: Translating Prolog to more idiomatic C#. In: *CI-CLOPS* 2004. pp. 59–70 (2004)
5. Demoen, B., Tarau, P.: jProlog home page (1997), <http://people.cs.kuleuven.be/~bart.demoen/PrologInJava/>
6. Eichberg, M.: Compiling Prolog to idiomatic Java. In: Gallagher, J.P., Gelfond, M. (eds.) *ICLP* 2011. pp. 84–94. Dagstuhl Publishing, Saarbrücken/Wadern (2011)
7. Filippou, J., Artikis, A., Skarlatidis, A., Paliouras, G.: A probabilistic logic programming event calculus (2012), <http://arxiv.org/abs/1204.1851>
8. Fisher, R.: CAVIAR test case scenarios. The EC funded project IST 2001 37540. (2007), <http://homepages.inf.ed.ac.uk/rbf/CAVIAR/>

9. Fujise, T., Chikayama, T., Rokusava, K., Nakase, A.: KLIC: A portable implementation of KL1. In: FGCS 1994. pp. 66–79. ICOT, Tokyo (Dec 1994)
10. Henderson, F., Somogyi, Z.: Compiling Mercury to high-level C code. In: CC 2002. Grenoble, France (2002)
11. Junior, J., Musse, S., Jung, C.: Crowd analysis using computer vision techniques. A survey. *IEEE Signal Processing Magazine* 27(5), 66–77 (Sep 2010)
12. Kim, I., Choi, H., Yi, K., Choi, J., Kong, S.: Intelligent visual surveillance—a survey. *International Journal of Control, Automation, and Systems* 8(5), 926–939 (2010)
13. Kuramochi, S.: KLIJava home page (1999), <http://www.ueda.info.waseda.ac.jp/~satoshi/klijava/klijava-e.html>
14. Machot, F., Kyamakya, K., Dieber, B., Rinner, B.: Real time complex event detection for resource-limited multimedia sensor networks. In: Workshop on Activity monitoring by multi-camera surveillance systems (AMMCSS). pp. 468–473 (2011)
15. 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. Paris, France (Sep 1999), <http://www.cplire.ru/Lab144/paris.pdf>
16. Morozov, A.A.: On semantic link between logic, object-oriented, functional, and constraint programming. In: MultiCPL 2002. Ithaca, NY, USA (Sep 2002), <http://www.cplire.ru/Lab144/multicpl.pdf>
17. Morozov, A.A.: Logic object-oriented model of asynchronous concurrent computations. *Pattern Recognition and Image Analysis* 13(4), 640–649 (2003), <http://www.cplire.ru/Lab144/pria640.pdf>
18. 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. pp. 1–15. Porto, Portugal (Sep 2007), <http://www.cplire.ru/Lab144/ciclops07.pdf>
19. Morozov, A.A.: Visual logic programming based on the SADT diagrams. In: Dahl, V., Niemela, I. (eds.) ICLP 2007. pp. 436–437. LNCS 4670, Springer, Heidelberg (2007)
20. Morozov, A.A.: Actor Prolog to Java translation (in Russian). In: IIP-9. pp. 696–698. Torus Press Moscow, Budva, Montenegro (2012)
21. Morozov, A.A.: A GitHub repository containing source codes of Actor Prolog built-in classes (including the Vision package) (2014), <https://github.com/Morozov2012/actor-prolog-java-library>
22. Morozov, A.A., Vaish, A., Polupanov, A.F., Antciperov, V.E., Lychkov, I.I., Al-fimtsev, A.N., Deviatkov, V.V.: Development of concurrent object-oriented logic programming system to intelligent monitoring of anomalous human activities. In: Jr., A.C., Plantier, G., Schultz, T., Fred, A., Gamboa, H. (eds.) BIODEVICES 2014. pp. 53–62. SCITEPRESS (Mar 2014), <http://www.cplire.ru/Lab144/biodevices2014.pdf>
23. O’Hara, S.: VERSA—video event recognition for surveillance applications. M.S. thesis. University of Nebraska at Omaha. (2008)
24. Shet, V., Singh, M., Bahlmann, C., Ramesh, V., Neumann, J., Davis, L.: Predicate logic based image grammars for complex pattern recognition. *International Journal of Computer Vision* 93(2), 141–161 (Jun 2011)
25. Tarau, P.: The BinProlog experience: Architecture and implementation choices for continuation passing Prolog and first-class logic engines. *Theory and Practice of Logic Programming* 12(1–2), 97–126 (2012)