A Virtual Machine for Low-Level Video Processing in Actor Prolog

Alexei A Morozov, Olga S Sushkova

Kotel'nikov Institute of Radio Engineering and Electronics of RAS, Mokhovaya 11-7, Moscow, Russia, 125009

Abstract. A virtual machine for low-level video processing in the Actor Prolog object-oriented logic language was developed. The principle of operation of this machine is the following one: (1) The machine keeps a sequence of commands of the low-level video processing. This sequence of commands is to be applied to every frame of the video. The loading of these commands into the machine is performed using predicates of the *VideoProcessingMachine* built-in class. (2) The machine keeps internal data arrays that are related to various sub-stages of the low-level video processing. Currently, the following sub-stages of the processing are implemented in the machine: preprocessing of the frame; processing of the frame in the pixel representation; selection of and processing the foreground pixels in the frame; extraction and tracing the blobs in the sequence of the frames. (3) The machine supports a stack of masks of foreground pixels. This stack enables the processing of different groups of blobs using different methods of image processing. (4) The result of the processing the frame of the video is a set of graphs that contain information about the movements of the blobs as well as other attributes of the blobs in the video scene during given time interval.

1. Introduction

Early research projects in the area of image processing and analysis by the means of the logic languages were linked with the appearance of efficient implementations of the Prolog language several decades ago [1–3]. These projects implemented technical possibilities to connect the Prolog language with the hardware and/or software libraries for image processing. One can enumerate the following reasons that make the logic programming attractive for image processing; these ideas were implemented in various combination in the projects [1, 2, 4–9]:

(i) The formulae of the first order logic (that is, the formulae containing the object variables) are convenient for the declarative description of the temporary and spatial relations between the elements of the image to be analyzed.

(ii) The logical formulae are convenient for the creation of composite descriptions of the graphic scenes and objects on the base of simple descriptions. These composite descriptions can be quite complex, but in any way, the human can investigate and understand them.

(iii) The logical formulae can be generated automatically by the means of the inductive logic programming [10] and other techniques; this idea is useful for solving various problems in the area of machine learning.

One of the basic problems that complicate the practical implementation of these ideas is a contradiction between the enumeration-of-possibilities principle implemented in the logical

inference and the necessity of the fast and efficient processing the graphics data. For instance, the backtracking mechanism that is typical for the logic languages recovers the previous state of the variables and composite data structures in the logic program, when the program has to investigate a new path of the execution. This principle is simple and efficient from the implementation point of view. However, the application of this principle to the image analysis causes a necessity to store copies of the image in the memory during the processing. This is necessary to implement the recovery of previous states of the images during the backtracking of the logic program; after the recovery, the program can try another way of the image processing.

This problem is solved in different ways in various logic programming projects. It is obvious, that the simplest solution of the problem is to avoid the storing the copies of the images during the logical inference. This approach was implemented in an early project [2]. Note that avoiding the image storing may violate or may not violate the declarative semantics of the logic programs. One can consider the built-in predicates that modify the images as always true statements; in this case, the image modification can be considered as a side effect of the execution of these true predicates. One can name this idea as the insertion of *out-of-logic operations* to the logic language. The problem of this approach is in that it can violate the declarative semantics of the logic program if one applies image processing operations that read (but not only modify) the state of the image, for instance, the intensity and/or colors of pixels. The declarative semantics will be violated if the logic program depends on the state of the pixels that it can modify. It is clear that most of the practical logic programs are of this kind.

The capability of the backtracking the logic programs that operate with the images is probably the main advantage of the connection of the logic programming with the image processing. In particular, it was the reason of why the team of the project [2] has implemented the idea of the backtracking the state of the images after several years [3,11].

Later on, the same problem has arisen in the area of intelligent visual surveillance systems [12]. The main reason of using logical rules in these systems was the possibility to declare complex situations and events (such as a stealing, an attempt to deceive an access control system, etc.) in a compact and understandable way. However, the problem of logical inference on the images was complicated by the fact that the intelligent video surveillance systems imply the real-time processing the sequence of frames, but not single images.

It is interesting that the complication of the problem of logical processing became a stimulus for the development of more complex, but more realistic approaches to the solution of the problem. The necessity of the real-time image processing was an essential reason for the separation of the low-level and high-level stages of the image processing. For example, in the experimental system [12], all preliminary video processing operations (including background subtraction and extraction of blobs) was moved out of the logical inference. In this system, the results of the low-level image processing are converted to the atomic formulae (so-called facts of the logic program) and analyzed using the convenient backtracking mechanism of the Prolog language.

A similar approach was applied in the project of the logic programming the intelligent video surveillance systems based on the Actor Prolog logic language [13–22]. In the framework of this project, one distinguishes the low-level video processing stage (that is, background subtraction, extraction of blobs, tracing the blobs, detection of intersections between the tracks of the blobs, etc.) and the high-level stage of the video processing. The former stage is implemented using specialized built-in classes of the logic language that were written in Java. The latter stage is implemented by the rules written in Actor Prolog.

Initially, the *ImageSubtractor* built-in class was developed and implemented in the Actor Prolog language that solves major tasks of the low-level video analysis:

(i) The preprocessing of the video including the 2D Gaussian filtering, 2D rank filtering, and background subtraction based on a simple Gaussian model of the background.

(ii) The extraction of the blobs and estimation of the coordinates and instantaneous velocities of the blobs.

(iii) The computation of the trajectories of the blob movements. The trajectory of the blob is separated into segments; the intersection of the trajectories is considered as the beginning and/or endpoints of the segments.

(iv) The detection and elimination of the trajectories of static and slow objects. It is useful to exclude slow objects from the analysis in some applications to decrease the search space and the size of the data structures describing the video scene.

(v) The computation of the graphs of the blob trajectories and the transformation of them to the terms of the Actor Prolog logic language. These data structures contain information about the coordinates and velocities of the blobs as well as values of special statistical metrics that describe the softness of the movements of the blobs [13, 19].

The means of the *ImageSubtractor* built-in class are enough for solving simple tasks of the intelligent visual surveillance. However, this class provides a strictly defined sequence of low-level image processing operations that is not a kind of universal and flexible approach. The experience of the logic programming the video analysis has revealed the necessity of more flexible means of the low-level image processing. In particular, one needs to form special sequences of low-level operations for solving special video analysis problems. For instance, in the applications of the monitoring the laboratory animals, it is necessary to operate simultaneously with blobs of various kinds; each type of the blobs requires the application of different methods of image preprocessing and extraction of the blobs.

In this paper, a new built-in class *VideoProcessingMachine* is considered that was developed in the Actor Prolog logic language for solving the problems of the low-level video processing described above. This class implements a virtual machine for the low-level video processing. In the second section of the paper, the architecture and basic principles of operation of this machine are considered. In the third section of the paper, an example of the logic program is discussed that uses the *VideoProcessingMachine* built-in class for the low-level video processing.

2. The Architecture of the Low-Level Video Processing Virtual Machine

The *VideoProcessingMachine* built-in class implements a kind of a virtual machine for the low-level video processing. The principle of operation of this machine is the following one:

(i) The machine keeps a sequence of commands of the low-level video processing. This sequence of commands is to be applied to every frame of the video. The loading of these commands into the machine is performed using special predicates of the *VideoProcessingMachine* class. After the loading of the commands, the sequence is repeated automatically for every incoming frame until the sequence will be updated by the programmer.

(ii) The machine keeps internal data arrays that are related to various sub-stages of the low-level video processing. Now the following sub-stages of the processing are implemented in the machine: preprocessing of the frame; processing of the frame in the pixel (matrix) representation; selection and processing of the foreground pixels in the frame; extraction and tracing the blobs in the sequence of the frames. Creation and conversion of the internal data arrays are implemented automatically in the course of the processing of the frame.

(iii) The machine supports a stack of masks of foreground pixels. This stack enables processing of different groups of blobs using different methods of image processing and blob extraction.

(iv) The result of the processing every frame of the video is a set of lists/graphs that contain information about the movements and other attributes of the blobs in the video scene during given time interval.

Table 1. The architecture of the Video Processing Machine.

Sub-stage of the processing	Commands of the virtual machine	Internal arrays of the machine
Preprocessing of the frame	Extract a sub-image; Resize the image; Gaussian filtering; etc.	An image (the *BufferedImage* class of the Java language)
Processing of the frame in the pixel representation	Select a channel in HSB/RGB space or the grayscale channel; Smooth the image; Compute gradients; Normalize pixels; etc.	A matrix of pixels (the *int[]* data type of the Java language)
Selection and processing of the foreground pixels in the frame	Push a new mask of foreground pixels into the stack; Pop the top mask from the stack; Add to the mask pixels corresponding to given conditions; Eliminate from the mask pixels corresponding to given conditions; Subtract background using given algorithm; Implement rank filtering of the foreground mask; Erode the mask of the foreground pixels; Dilate the mask; etc.	A mask of the foreground pixels (the *boolean[]* data type of the Java language)
Extraction and tracing the blobs in the frames	Extract blobs using given algorithm; Fill the blobs; Select blobs corresponding to given conditions; Trace the movements of the blobs; Compute color histograms of the blobs; etc.	Lists of coordinates and other attributes of the blobs
Additional processing the tracks of the blobs	Select tracks corresponding to given conditions; etc.	A list of tracks

The list of the sub-stages of the low-level video processing implemented in the machine as well as the list of corresponding commands and internal data arrays are shown in Table 1.

With relation to the declarative semantics of the logic language, the virtual machine is a kind of *out-of-logic means* of the language, that is, the built-in predicates of the *VideoProcessingMachine* class are the true statements and have no direct impact on the execution of the logic program. The creation and implementation of the sequences of the low-level video processing commands is a side effect of these predicates. Surely this is not an acceptable explanation of what is the declarative semantics of the logic program that processes a video. Nevertheless one can guarantee the mathematical soundness and completeness of separate sections of the video processing logic program when it is necessary. For instance, the programmer can divide the logic program into two parallel communicating processes (two logic programs). The first process has to create a sequence of low-level video processing commands and control the virtual machine. The second process has to analyze graphs computed by the virtual machine. The system of these two processes has no declarative semantics itself, but each separate process has a clear declarative semantics and the programmer can guarantee the soundness and completeness of each separate logic program. Note that the programmer has to make additional efforts to avoid infinite looping in the logic program to guarantee the completeness of the program.

3. An Example of a Logic Program Analyzing a Video

Let us consider an example of a logic program written in Actor Prolog that extracts blobs of given types in a video. The program has to create a dialog window to demonstrate the sequence of the frames and results of the detection of the blobs of the following types: electroencephalographic (EEG) cap on the head of a laboratory rat that connects the head of the rat with EEG cable, green and blue objects placed in the cage in the course of a laboratory test (see Figure 1).

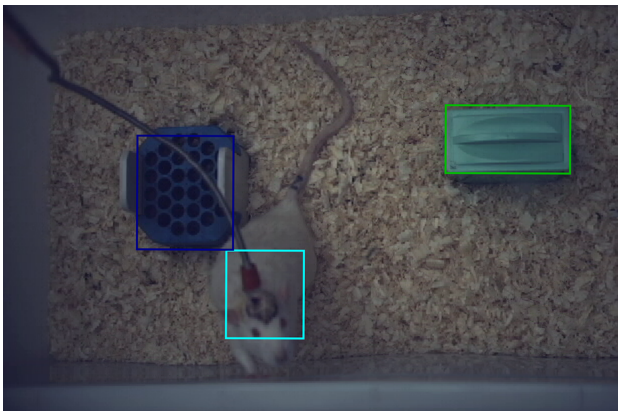

Figure 1. A laboratory rat investigates new objects in the cage. The logic program has detected three blobs in the video: EEG cap that connects the head of the rat with EEG cable, a green object, and a blue object placed in the cage. The data are coming from the Institute of Higher Nervous Activity and Neurophysiology of RAS.

Let us use the *FFmpeg* built-in class that links Actor Prolog with the FFmpeg open library to read the video file. The main class of the logic program is a descendant of the *FFmpeg* class. The *name* slot of the class contains the name of the video file. Besides, the *Main* class includes auxiliary slots *vpm*, *dialog*, *graphics_window*, *image1*, and *image2*:

```
class 'Main' (specialized 'FFmpeg'):
constant:
     name = "rats.avi";
internal:
     vpm       = ('VideoProcessingMachine');
     dialog    = ('DemoPanel',
                    graphics_window);
     graphics_window = ('Canvas2D');
     image1    = ('BufferedImage');
     image2    = ('BufferedImage');
```

The *vpm* slot contains an instance of the *VideoProcessingMachine* class. The *dialog* class contains a dialog window. The *graphics_window* slot contains a graphics window intended for displaying video frames and blobs. The *image1* and *image2* slots contain instances of the *BufferedImage* built-in class that is intended for storing and transfer of the graphics data arrays.

The goal statement of the logic program (*goal*) loads commands to the instance of the *VideoProcessingMachine* class, then opens the dialog window, and initiates reading the video file:

```
CLAUSES:
goal:-!,
     store_VPM_program,
     dialog ? maximize,
     start.
```

The *store_VPM_program* predicate loads a sequence of commands to the virtual machine using special predicates of the *VideoProcessingMachine* built-in class. Note that the program uses different low-level processing algorithms for extracting blobs of different types. The coordinates of EEG cap and the objects placed in the cage are detected using different sub-spaces in the Hue-Saturation-Brightness (HSB) color space.

```
store_VPM_program:-
     -- Suspend the video processing and delete all commands:
     vpm ? suspend_processing,
     vpm ? retract_all_instructions,
     -- Assign the size of blob borders:
     vpm ? blb_set_blob_borders(15,15),
     -- Create a new foreground mask:
     vpm ? msk_push_foreground,
     -- Select pixels corresponding to given interval of hue:
     vpm ? msk_select_foreground('HUE',185,245),
     -- Select pixels corresponding to given interval of saturation:
     vpm ? msk_select_foreground('SATURATION',50,255),
     -- Select pixels corresponding to given interval of brightness:
     vpm ? msk_select_foreground('BRIGHTNESS',0,150),
     -- Erode the mask by 2 pixels:
     vpm ? msk_erode_foreground(2),
     -- Dilate the mask by 2 pixels:
     vpm ? msk_dilate_foreground(2),
     -- Extract blobs of the cap type:
     vpm ? blb_extract_blobs('cap','TWO_PASS_BLOB_EXTRACTION'),
     -- Select the biggest blob:
     vpm ? blb_select_superior_blob('FOREGROUND_AREA'),
     -- Fill gaps in the blobs:
     vpm ? blb_fill_blobs,
     -- Pop the mask from the stack:
     vpm ? msk_pop_foreground,
     -- Assign the size of blob borders:
     vpm ? blb_set_blob_borders(1,1),
     -- Create a new foreground mask:
     vpm ? msk_push_foreground,
     -- Select pixels corresponding to given interval of hue:
     vpm ? msk_select_foreground('HUE',111,138),
     -- Select pixels corresponding to given interval of saturation:
     vpm ? msk_select_foreground('SATURATION',50,255),
     -- Erode the mask by 2 pixels:
     vpm ? msk_erode_foreground(2),
     -- Dilate the mask by 2 pixels:
     vpm ? msk_dilate_foreground(2),
     -- Extract blobs of the green_object type:
     vpm ? blb_extract_blobs('green_object','TWO_PASS_BLOB_EXTRACTION'),
     -- Select the biggest blob:
     vpm ? blb_select_superior_blob('FOREGROUND_AREA'),
     -- Fill gaps in the blobs:
     vpm ? blb_fill_blobs,
     -- Pop the mask from the stack:
     vpm ? msk_pop_foreground,
     -- Create a new foreground mask:
     vpm ? msk_push_foreground,
     -- Select pixels corresponding to given interval of hue:
```

```
    vpm ? msk_select_foreground('HUE',147,183),
    -- Select pixels corresponding to given interval of saturation:
    vpm ? msk_select_foreground('SATURATION',103,255),
    -- Erode the mask by 2 pixels:
    vpm ? msk_erode_foreground(2),
    -- Dilate the mask by 2 pixels:
    vpm ? msk_dilate_foreground(2),
    -- Extract blobs of the blue_object type:
    vpm ? blb_extract_blobs('blue_object','TWO_PASS_BLOB_EXTRACTION'),
    -- Select the biggest blob:
    vpm ? blb_select_superior_blob('FOREGROUND_AREA'),
    -- Fill gaps in the blobs:
    vpm ? blb_fill_blobs,
    -- Pop the mask from the stack:
    vpm ? msk_pop_foreground,
    -- Resume the video processing:
    vpm ? process_now.
```

When a new frame is received from the video file, the *frame_obtained* predicate is invoked automatically in the logic program. The logic program informs the *FFmpeg* built-in class using the *commit* predicate that it is going to process the frame. Then it loads the content of the frame into the *image1* object using the *get_recent_image* predicate. The *process_realtime_frame* predicate of the *VideoProcessingMachine* class is used to transfer the frame to the Video Processing Machine. Note that the arrays of the video data remain inside the built-in classes all the time during the data processing that is necessary to ensure the effectiveness of the low-level image processing.

```
frame_obtained:-
    commit,!,
    get_recent_image(image1),
    vpm ? process_realtime_frame(image1),
    draw_scene.
```

After that, the *draw_scene* predicate is called. This predicate accepts a list of blobs from the Video Processing Machine and displays the blobs in the dialog window. The predicate informs the Video Processing Machine by the *commit* predicate that it needs data structures describing the blobs in the current video frame. Then it accepts the content of the video frame using the *get_recent_image* predicate and the list of the blobs using the *get_blobs* predicate. The image and blobs are displayed in *graphics_window* using predicates of the *Canvas2D* built-in class:

```
draw_scene:-
    vpm ? commit,
    vpm ? get_recent_image(image2),
    image2 ? get_size_in_pixels(IW,IH),
    vpm ? get_blobs(BlobList),
    graphics_window ? suspend_redrawing,
    graphics_window ? clear,
    graphics_window ? draw_image(image2,0,0,1,1),
    draw_blob_list(BlobList,IW,IH),
    graphics_window ? draw_now.
```

The list of the blobs is unrolled by the *draw_blob_list* predicate and each blob is drawn in the window as a colored rectangle:

```
draw_blob_list([Blob|Rest],IW,IH):-
    draw_blob(Blob,IW,IH),
    draw_blob_list(Rest,IW,IH).
draw_blob_list([],_,_).
```

The blob is described using an underdetermined set [23] of Actor Prolog that contains information about the type, coordinates, width, height, and other attributes of the blob:

```
draw_blob(Blob,IW,IH):-
     Blob == {type:Type,x:X0,y:Y0,width:W1,height:H1|_},
     X2== (X0 - W1/2) / IW,
     Y2== (Y0 - H1/2) / IH,
     W2== W1 / IW,
     H2== H1 / IH,
     select_blob_color(Type,Color),
     graphics_window ? set_pen({color:Color,lineWidth:3}),
     graphics_window ? draw_rectangle(X2,Y2,W2,H2),
     fail.
draw_blob(_,_,_).
```

The *select_blob_color* predicate defines the colors of the rectangles to be used for drawing the blobs:

```
select_blob_color('cap','Cyan').
select_blob_color('green_object','DkGreen').
select_blob_color('blue_object','Navy').
```

This plain example illustrates both the low-level video processing and the high-level video processing, as well as the usage of the Video Processing Machine for extraction and handling blobs of various types in the video.

4. Conclusion

A new method of the low-level video analysis in the logic language is developed and implemented. The advantage of this method is in that one can form specialized sequences of low-level video processing commands for various video analysis applications. One can operate with the blobs of different kinds simultaneously, that is, apply different algorithms of image preprocessing and blob detection to the blobs of different kinds. The developed method is implemented in the *VideoProcessingMachine* built-in class of the Actor Prolog object-oriented logic language. The software is freely available on the Web Site [22]. The source codes of the *VideoProcessingMachine* built-in class are published in the GitHub repository [24].

Acknowledgments

Authors are grateful to IHNA and NPh RAS for the experimental video data. This research is supported by the Russian Foundation for Basic Research (grant number 16-29-09626-ofi-m).

References

[1] Bell B and Pau L 1990 *IEEE Transactions on Pattern Analysis and Machine Intelligence* **12** 913–917

[2] Batchelor B G 1991 *Intelligent Image Processing in Prolog* (London: Springer-Verlag)

[3] Jones A C 1995 *The ALP Newsletter* **8/4**

[4] Bell B and Pau L 1992 *Pattern Recognition Letters* **13** 279–290

[5] Jones A C and Batchelor B G 1994 A software architecture for intelligent image processing using Prolog *Proc. Intelligent Robots & Computer Vision XIII* ed Casasent D (Bellingham WA: Int. Society for Optical Engineering) pp 178–188

[6] Stafford-Fraser Q and Robinson P 1996 BrightBoard: A video-augmented environment *CHI Electronic Proceedings*

[7] Esposito F, Malerba D and Lisi F A 2000 *Journal of Intelligent Information Systems* **14** 175–198

[8] Batchelor B G and Whelan P F 1997 *Intelligent Vision Systems for Industry* (London: Springer)

[9] Cacko A and Iwanowski M 2015 Evaluating the mutual position of objects on the visual scene using morphological processing and reasoning *Image Processing & Communications Challenges* (*Advances in Intelligent Systems and Computing* vol 313) ed Choraś R S (Heidelberg: Springer International Publishing) pp 13–20

[10] Drobics M, Bodenhofer U and Klement E P 2003 *International Journal of Approximate Reasoning* **32** 131–152
[11] Jones A C and Batchelor B G 1995 Implementing full backtracking facilities for Prolog-based image processing *Proc. Machine Vision Applications, Architectures and Systems Integration IV* (Bellingham WA: Int. Society for Optical Engineering)
[12] Shet V, Harwood D and Davis L 2005 VidMAP: Video monitoring of activity with Prolog *AVSS 2005* (IEEE) pp 224–229
[13] Morozov A A and Sushkova O S 2016 *Computer Optics* **40** 947–957
[14] Morozov A A, Sushkova O S and Polupanov A F 2017 Object-oriented logic programming of 3D intelligent video surveillance: The problem statement *2017 IEEE 26th International Symposium on Industrial Electronics (ISIE), Edinburgh, United Kingdom, 2017* (Washington: IEEE Xplore Digital Library) pp 1631–1636 http://ieeexplore.ieee.org/document/8001491
[15] Morozov A A, Sushkova O S and Polupanov A F 2017 Towards the distributed logic programming of intelligent visual surveillance applications *Advances in Soft Computing: 15th Mexican International Conference on Artificial Intelligence, MICAI 2016, Cancun, Mexico, Proceedings, Part II* ed Pichardo-Lagunas O and Miranda-Jimenez S (Cham: Springer International Publishing) pp 42–53
[16] Morozov A A 2015 *Pattern Recognition and Image Analysis* **25** 481–492
[17] Morozov A A, Sushkova O S and Polupanov A F 2015 An approach to the intelligent monitoring of anomalous human behaviour based on the Actor Prolog object-oriented logic language *RuleML 2015 DC and Challenge* ed Bassiliades N, Fodor P, Giurca A, Gottlob G, Kliegr T, Nalepa G, Palmirani M, Paschke A, Proctor M, Roman D, Sadri F and Stojanovic N (Berlin: CEUR) URL https://www.csw.inf.fu-berlin.de/ruleml2015-ceur
[18] Morozov A A, Sushkova O S and Polupanov A F 2015 A translator of Actor Prolog to Java *RuleML 2015 DC and Challenge* ed Bassiliades N, Fodor P, Giurca A, Gottlob G, Kliegr T, Nalepa G, Palmirani M, Paschke A, Proctor M, Roman D, Sadri F and Stojanovic N (Berlin: CEUR)
[19] Morozov A A and Polupanov A F 2015 Development of the logic programming approach to the intelligent monitoring of anomalous human behaviour *OGRW2014* ed Paulus D, Fuchs C and Droege D (Koblenz: University of Koblenz-Landau) pp 82–85
[20] Morozov A A and Polupanov A F 2014 Intelligent visual surveillance logic programming: Implementation issues *CICLOPS-WLPE 2014* (*Aachener Informatik Berichte* no AIB-2014-09) ed Ströder T and Swift T (RWTH Aachen University) pp 31–45
[21] Morozov A A, Vaish A, Polupanov A F, Antciperov V E, Lychkov I I, Alfimtsev A N and Deviatkov V V 2015 Development of concurrent object-oriented logic programming platform for the intelligent monitoring of anomalous human activities *BIOSTEC 2014* (*CCIS* vol 511) ed Plantier G, Schultz T, Fred A and Gamboa H (Springer) pp 82–97
[22] Morozov A A and Sushkova O S 2018 The intelligent visual surveillance logic programming Web Site URL http://www.fullvision.ru
[23] Morozov A A 1999 Actor Prolog: an object-oriented language with the classical declarative semantics *IDL 1999* ed Sagonas K and Tarau P (Paris, France) pp 39–53
[24] Morozov A A 2018 A GitHub repository containing source codes of Actor Prolog built-in classes URL https://github.com/Morozov2012/actor-prolog-java-library