A Translator of Actor Prolog to Java

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

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

Abstract. Actor Prolog is a concurrent object-oriented logic language developed in [1]. We demonstrate a state-of-the-art translator of Actor Prolog to Java developed in the framework of the Intelligent Visual Surveillance Logic Programming project [2]. The translator implements a set of high-level and low-level code optimization methods and generates a kind of the idiomatic (i.e., well-readable) source code in Java, that ensures a high speed, robustness, and openness of the executable code. Some applications of the Actor Prolog to Java translator are demonstrated, in particular, the real-time intelligent video surveillance, Actor Prolog with Java3D linking, and logic programming of Java applets.

1 Introduction

Industrial applications of a logic programming language impose a number of strong and contradictory requirements to the logic programming system, i.e., speed, robustness, and openness of the executable code. These requirements are contradictory, since a considerable code optimization implies application of complex compilation algorithms and a low-level code generation that may be the sources of difficult-to-locate errors, memory leaks, and unstable operation of the executable code. Even if the compiler is well-debugged the continuous development of built-in classes and libraries is a constant potential source of such errors. Thus, there is a fundamental contradiction between the openness and the optimization of the programming system.

In applications we use a compilation of the Actor Prolog language (see [3,4,5,6]) to Java because we are sure that modern processors are fast enough to neglect the speed of the executable code for the sake of robustness, readability, and openness of the logic programs. Moreover, using an industrial Java virtual machine as a basis for a logic programming system ensures its flexibility and fast adaptation to the new operating systems and processor architectures.

The Actor Prolog language is significantly different from the conventional Clocksin&Mellish Prolog. In fact, Turbo-Prolog style domain and predicate declarations of Actor Prolog are of importance for the industrial application programming and is helpful for the executable code optimization, but, on the other hand, object-oriented features and supporting concurrent programming make the translation to be a non-trivial problem.

The compilation schema of the Actor Prolog language is described in Section 2. A brief comparison with other approaches to the logic languages compilation is represented in Section 3. Some applications of the Actor Prolog to Java translator are described in Section 4.

2 The Compilation Schema

The state-of-the-art compilation schema of the Actor Prolog system includes the following steps [6]:

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.
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. In particular, this information is used to eliminate all unused predicates from the executable code.
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 a mutual influence of the clauses / predicates. This check is of the critical importance, since the imperative predicates, as a rule, constitute the main part of the program and the check ensures a deep optimization of these predicates. Clauses of the imperative predicates are translated to Java procedures directly.
2. The deterministic predicates are translated to Java procedures too. All clauses of one deterministic predicate correspond to a single Java procedure. Backtracking is implemented using a special kind of light-weight Java exceptions.
3. The 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.

The tail recursion optimization is implemented for recursive predicates. Recursive predicates are implemented using the *while* Java command. Note that Actor Prolog supports the explicit definition of ground / non-ground domains and the translator uses this information for a deep optimization of Java code.

Example 1. Let p be an imperative predicate that calls another imperative predicate q. The q predicate calls the built-in *writeln* predicate that outputs the text message "Hi, Berlin!":

```
p:-
        q.
q:-
        writeln("Hi, Berlin!").
```

The translator converts these predicates to procedures *impProcP_s*617_0 and *impProcQ_s*618_0. The names of these Java procedures are created automatically and contain information about determinism of the source predicates (the *imp* prefix), the names of the source predicates (P and Q), and the arity of the predicates (both predicates are of arity 0). The $s617$ and $s193$ infixes contain auxiliary information: the s letter indicates that a predicate is plain (but not an imitation of a function) and a number is a unique code that is necessary for unambiguous representation of predicates with non-English names.

```
public void impProcP_s617_0(ChoisePoint iX) {
        impProcQ_s618_0(iX);
}
public void impProcQ_s618_0(ChoisePoint iX) {
       impProcWriteln_s193_1_i1(
              iX,new PrologString("Hi, Berlin!"));
}
```

Note that the name of the Java procedure corresponding to the *writeln* predicate is coded in accordance with the same rules, but this predicate is of arity 1 and its name contains additional information about the argument: the letter i in the $i1$ postfix indicates that the argument is input and the number 1 in the postfix is a unique code of the domain (data type) of this argument. The text string is implemented in Java as an instance of a special class *PrologString*. The *iX* variable of the *ChoisePoint* class contains auxiliary information during the execution of the program.

Example 2. Let p be a deterministic predicate that calls a deterministic predicate q.

```
p:-
        q.
```

The translator converts the p predicate in the same way. The name of the corresponding Java procedure contains the *det* prefix. In addition, this Java procedure supports the *Backtracking* exception that is used for the implementation of backtracking during the execution of the program. Note that the *Backtracking* class implements a special kind of light-weight Java exceptions, that is, an instance of this class does not contain information about the current state of the stack frames for the current thread. Thus, the backtracking operates quite fast in Actor Prolog.

```
public void detProcP_s617_0(ChoisePoint iX)
          throws Backtracking {
     detProcQ_s618_0(iX);
}
```

Example 3. Let p be a non-deterministic predicate that calls the q non-deterministic predicate.

```
p:-
        q.
```

The translator converts the p predicate to the $NondetProcP\_s617\_0$ class using a standard method of continuation passing. The name of corresponding Java class contains the *Nondet* prefix and is created in accordance with the rules of Java procedure naming mentioned above. The constructor of the $NondetProcP\_s617\_0$ class has one argument of the Continuation class. This auxiliary class represents continuations.

The *execute* method of the class implements execution of the p predicate and can raise the *Backtracking* exception when the predicate fails. The procedure creates a new instance of the $NondetProcQ\_s618\_0$ class that implements the q non-deterministic predicate and calls the *execute* method of this object.

```
class NondetProcP_s617_0 extends Continuation {
     private Continuation c1;
     NondetProcP_s617_0(Continuation aC) {
          c0= aC;
     }
     public void execute(ChoisePoint iX)
               throws Backtracking {
          c1= new NondetProcQ_s618_0(c0);
          c1.execute(iX);
     }
}
```

Example 4. Let p be an imperative predicate that calls the q non-deterministic predicate. Let p contains two clauses. Let p contains a cut in the first clause after the call of the q predicate.

```
p:-
    q,!.
p:-
    writeln("P").
q:-
    writeln("Q").
```

The translator converts the p predicate to the $impProcP\_s694\_0$ Java procedure and the $And\_1\_1\_P\_s694\_0$ auxiliary Java class. Two clauses of the p predicate are implemented in Java using the $try - catch$ construct. The first clause is converted to the following commands:

1. Create an instance of the $And\_1\_1\_P\_s694\_0$ class.
2. Then create an instance of the $NondetProcQ\_s695\_0$ class corresponding to the q predicate.
3. Call the *execute* method of this continuation.

If the q predicate fails, the *Backtracking* exception is raised. This exception is to be processed by the *catch* construct. The further execution of the procedure depends on the state of the $newIx$ auxiliary variable that indicates whether the backtracking of the first clause is allowed. This variable contains an instance of the $ChoisePoint$ class that can be modified inside the q predicate by the cut operation. If the backtracking is allowed, the procedure frees the trail of the program and calls the $impProcWriteln\_s205\_1\_i1$ procedure that outputs the P text string. If the backtracking is disabled, the procedure raises the $ImperativeProcedureFailed$ run-time exception. Note that the translator checks the usage of cuts in the text of Actor Prolog program and ensures that this situation never occurs.

```
public void impProcP_s694_0(ChoisePoint iX) {
    Continuation c1;
    Continuation c2;
    ChoisePoint newIx;
    newIx= new ChoisePoint(iX);
    try {
        c1= new And_1_1_P_s694_0(c0,iX);
        c2= new NondetProcQ_s695_0(c1);
        c2.execute(newIx);
    } catch (Backtracking b1) {
        if (newIx.isEnabled()) {
            newIx.freeTrail();
            impProcWriteln_s205_1_i1(
                newIx,new PrologString("P"));
        } else {
            throw new ImperativeProcedureFailed();
        }
    }
}
```

The $And\_1\_1\_P\_s694\_0$ class implements further execution of the first clause of the p predicate after successful completion of the q predicate. Note that the translator converts the cut operation to the call of the *disable* method of the iX auxiliary variable that contains information about variable bindings and possible backtracking.

```
class And_1_1_P_s694_0 extends Continuation {
    private ChoisePoint pS;
    And_1_1_P_s694_0(
        Continuation aC, ChoisePoint aCP) {
        c0= aC;
        pS= aCP;
    }
    public void execute(ChoisePoint iX) throws Backtracking {
        iX.disable(pS);
        c0.execute(iX);
    }
}
```

The described compilation schema ensures a high performance of the executable code (see Table. 2). Deterministic and imperative predicates with ground arguments are optimized quite well (for instance, the NREV test demonstrates more than 100 millions lips). At the same time, the programs that exploit extensively non-deterministic predicates work slowly (QUERY). This is a fundamental disadvantage of the approach based on the continuation passing and translation to the high-level intermediate language, because it cannot handle possible runtime optimization of Prolog stacks. Arithmetical predicates work fast enough in Actor Prolog (PRIMES, QSORT, and TAK), but there is a possibility for better optimization of symbolic computations (DERIV, POLY_10).

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 [8] 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.

Our approach to Prolog and Java merging has the following advantages in comparison with an approach where a logic program and a Java program communicate through an interface as two separate black boxes (e.g., a Prolog program and a Java program exchange data through a Prolog-Java interface such as in SWI Prolog [7]):

1. *Portability of the programs.* The translator generates Java applets that can operate in any computer without preliminary installation of Actor Prolog; only Java is necessary.

Test	Iter. No.	Actor Prolog	SWI-Prolog
NREV	3,000,000	109,677,895 lips	15,792,155 lips
CRYPT	100,000	1.820880 ms	1.98979 ms
DERIV	10,000,000	0.055460 ms	0.0105815 ms
POLY_10	10,000	3.750600 ms	4.4257 ms
PRIMES	100,000	0.037340 ms	0.14196 ms
QSORT	1,000,000	0.043129 ms	0.063976 ms
QUEENS(9)	10,000	19.219600 ms	32.4248 ms
QUERY	10,000	3.135300 ms	0.4056 ms
TAK	10,000	3.913400 ms	11.1182 ms

Table 1. Prolog benchmark testing: Actor Prolog to Java translator vs. SWI-Prolog [7] v. 7.2.2, Intel Core i5-2410M, 2.30 GHz, Win7, 64-bit. Benchmarks time is measured in milliseconds per iteration. The LIPS abbreviation means the number of logical inferences per second.

2. *Reliability and stability of the programs.* The single language approach always ensures better reliability and stability of the application programs.
3. *Safety of the programs.* All Java features that ensure safety of the programs are available.
4. *Readability of the intermediate code.* The intermediate code can be easily inspected by a human if necessary.
5. *Availability of all Java means.* Java Internet protocols, Java2D, Java3D, and other Java libraries are available.
6. *Portability of the logic programming system.* The use of industrial virtual machine is a basis for quick adaptation to new operating systems and processor architectures.

The main disadvantages of the developed approach are the following:

1. The executable code is slow in comparison to the translation to C approach.
2. Only static optimization of the code is possible, because Java implements no advanced run-time optimization methods developed in the logic programming area.
3. The logic programming system depends on Java virtual machine.

3 Comparison with Other Approaches

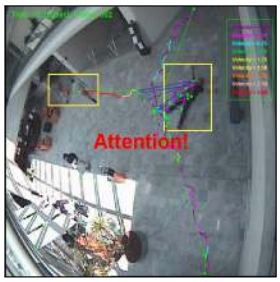
Application of a compilation schema based on C / C++ intermediate code generation (Mercury [9], KLIC [10], wamcc [11]) was recognized as an appropriate way to obtain maximal speed of the executable code. On the other hand, generation of Java intermediate code (Actor Prolog, PrologCafe [12], KLIJava [13], SAE-Prolog [14], jProlog [15]) 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 have chosen the second compilation schema to ensure robustness, readability, and openness of the executable code.

In contrast to conventional approaches, we use neither WAM (PrologCafe, wamcc) nor binarization of the logic program (jProlog, BinProlog [16]). The Actor Prolog compiler generates a kind of the idiomatic source code (SAE-Prolog, P# [17]), but in contrast to the SAE-Prolog project [14] we use domain and predicate declarations to process non-deterministic, deterministic, and imperative predicates in different ways. In contrast to the P# project [17] we implement non-idiomatic predicate calls from idiomatic predicates and vice versa.

4 Applications of the Translator

The main application and a test basis of the Actor Prolog to Java translator is the intelligent visual surveillance, that is, a real-time intelligent analysis of video streams and an intelligent monitoring of anomalous human behaviour [18,6,19,2]. The idea of the logic programming approach is in usage of logical rules for description and analysis of people activities. We solve the problem of anomalous human activity recognition using a logic program that describes a given scenario of complex people behaviour (see examples in Fig. 1).

(a) 

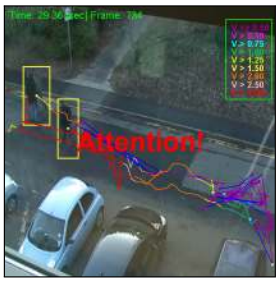
(b) 

Fig. 1. (a) An example of CAVIAR [20] video with a case of a street offence: one person attacks another. (b) An example of BEHAVE [21] video with a case of a street offence: one group attacks another. These cases of street offences are detected by logic programs. All probable participants of conflicts are marked by yellow rectangles. The tracks are designated by lines.

Let us consider a short fragment of a logic program to demonstrate some issues of logical description of video scenes. The *is_a_running_person* predicate checks whether a term E corresponds to fuzzy definition of a running person:

```
is_a_running_person(E):-
     E == {    frame1:T1,
               frame2:T2,
               mean_velocity:V,
               wr2_mean:M,
```

```
            wr2_skewness:S,
            mean_standardized_area:A,
            wr2_cardinality:C|_},
    is_a_fast_object(T1,T2,V),
    fast_object_is_a_runner(A,M,S,C).
```

The E term is an underdetermined set [3] that is a special data structure introduced in Actor Prolog. This term represents an unordered set of named attributes of a track segment of an object in a video scene: numbers of the first and the last frames ($frame1$, $frame2$); the average speed of the object in this segment of the track ($mean\_velocity$); a number of statistical metrics [19] describing the motion of the object ($wr2\_mean$, $wr2\_skewness$, $mean\_standardized\_area$, and $wr2\_cardinality$). Note that in the Actor Prolog language, the == operator corresponds to the = ordinary unification of the standard Prolog. In accordance with the rule, the E term corresponds to the running person if and only if:

1. It is recognised as a fast object, i.e., the speed and the length of the track segment satisfy the fuzzy definition of the fast object (see the rule below).
2. The values of given statistical metrics satisfy the fuzzy definition of the running pedestrian.

The values of fuzzy thresholds used in the rules were computed on the basis of BEHAVE [21] video samples.

```
is_a_fast_object(T1,T2,V):-
    M1== ?fuzzy_metrics(V,1.7,0.7),
    D== (T2 - T1) / 25,
    M2== ?fuzzy_metrics(D,0.5,0.25),
    M1 * M2 >= 0.5.
fast_object_is_a_runner(A,M,S,C):-
    MC== ?fuzzy_metrics(C,7,2),
    MA== 1 - ?fuzzy_metrics(A,2.75,0.75),
    MM== 1 - ?fuzzy_metrics(M,0.49,0.10),
    MS== ?fuzzy_metrics(S,0.25,1.00),
    MA * MM * MS * MC >= 0.1.
```

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)).
```

Another area of application of the translator is in linking Actor Prolog with Java3D [22]. Java3D is a scene graph based 3D application programming interface / graphics library. Recently a group of enthusiasts has ported this library to the JOGL graphics platform [23], that is, it uses only JOGL on all computer platforms and a Java applet can use Java3D capabilities on any computer without preliminary installation of the Java3D library. We have developed a built-in class of the Actor Prolog language that is a wrapper of Java3D [8] and provided a set of examples of logic programming 3D graphics (see Fig. 2).

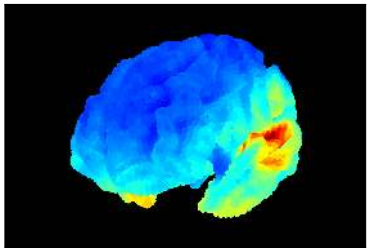
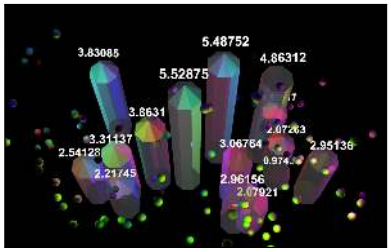
(a)  (b) 

Fig. 2. Examples of the scientific visualization based on the Actor Prolog language and the Java3D graphics: (a) Visualization of results of a neurophysiologic experiment. (b) Creation of dynamic 3D diagrams.

An advantage of the translation to Java is in the access to all Java security features including protection of Java applets. Recent versions of the Actor Prolog system ensure automatic signing of created JAR files, that facilitates logic programming Web applications based on Java applets (see Fig. 3).

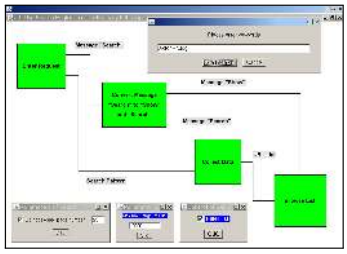
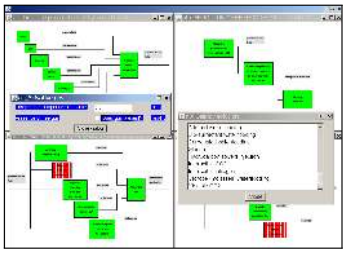
(a)  (b) 

Fig. 3. Examples of Java applets generated by Actor Prolog: (a) A Web agent that communicates with the Rambler search engine. (b) An expert system for selection of oil production methods with a SADT [24] based user interface.

An open source Java library of Actor Prolog built-in classes including the video processing features and Java2D / Java3D wrappers is published in the GitHub repository [8].

5 Conclusions

A translator of Actor Prolog to Java was developed. The state-of-the-art compilation schema of the Actor Prolog system includes a type check, a determinism check, and a global flow analysis. This compilation schema ensures a high performance of the executable code. We use a compilation from the Actor Prolog language to Java, because, from our point of view, using an industrial Java virtual machine as a basis for the logic programming system ensures its stability, flexibility, and quick adaptation to new operating systems and processor architectures. The open source Java library of Actor Prolog built-in classes is published in GitHub [8]. Application domains of the translator include but are not limited to the real-time intelligent monitoring of anomalous people activities, the logical description and analysis of people behaviour (see the Web Site [2]), 3D scientific visualization, and logic programming Web applications.

Acknowledgements

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. Morozov, A.A.: Actor Prolog Web Site. [Online] Available from: http://www.cplire.ru/Lab144 (2004)
2. Morozov, A.A., Sushkova, O.S.: The intelligent visual surveillance logic programming Web Site. [Online] Available from: http://www.fullvision.ru/actor_prolog/ (2014)
3. Morozov, A.A.: Actor Prolog: an object-oriented language with the classical declarative semantics. In Sagonas, K., Tarau, P., eds.: IDL 1999, Paris, France (1999) 39–53
4. Morozov, A.A.: Logic object-oriented model of asynchronous concurrent computations. Pattern Recognition and Image Analysis **13** (2003) 640–649
5. Morozov, A.A.: Operational approach to the modified reasoning, based on the concept of repeated proving and logical actors. In Salvador Abreu, V.S.C., ed.: CICLOPS 2007, Porto, Portugal (2007) 1–15
6. Morozov, A.A., Polupanov, A.F.: Intelligent visual surveillance logic programming: Implementation issues. In Ströder, T., Swift, T., eds.: CICLOPS-WLPE 2014. Number AIB-2014-09 in Aachener Informatik Berichte, RWTH Aachen University (2014) 31–45
7. Wielemaker, J., Schrijvers, T., Triska, M., Lager, T.: SWI-Prolog. Theory and Practice of Logic Programming **12** (2012) 67–96
8. Morozov, A.A.: A GitHub repository containing source codes of Actor Prolog built-in classes. [Online] Available from: https://github.com/Morozov2012/actor-prolog-java-library (2014)
9. Henderson, F., Somogyi, Z.: Compiling Mercury to high-level C code. In: CC 2002, Grenoble, France (2002)

10. Fujise, T., Chikayama, T., Rokusava, K., Nakase, A.: KLIC: A portable implementation of KL1. In: FGCS 1994, Tokyo, ICOT (1994) 66–79
11. Codognet, P., Diaz, D.: wamcc: Compiling Prolog to C. In Sterling, L., ed.: ICLP 1995, MIT Press (1995) 317–331
12. 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. LNAI 4369, Heidelberg, Springer (2006) 1–11
13. Kuramochi, S.: KLIJava home page. [Online] Available from: http://www.ueda.info.waseda.ac.jp/~satoshi/klijava/klijava-e.html (1999)
14. Eichberg, M.: Compiling Prolog to idiomatic Java. In Gallagher, J.P., Gelfond, M., eds.: ICLP 2011, Saarbrücken/Wadern, Dagstuhl Publishing (2011) 84–94
15. Demoen, B., Tarau, P.: jProlog home page. [Online] Available from: http://people.cs.kuleuven.be/~bart.demoen/PrologInJava/ (1997)
16. 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** (2012) 97–126
17. Cook, J.J.: Optimizing P#: Translating Prolog to more idiomatic C#. In: CICLOPS 2004. (2004) 59–70
18. Morozov, A.A., Vaish, A., Polupanov, A.F., Antciperov, V.E., Lychkov, I.I., Alfimtsev, A.N., Deviatkov, V.V.: Development of concurrent object-oriented logic programming system to intelligent monitoring of anomalous human activities. In Jr., A.C., Plantier, G., Schultz, T., Fred, A., Gamboa, H., eds.: BIODEVICES 2014, SCITEPRESS (2014) 53–62
19. Morozov, A.A., Polupanov, A.F.: Development of the logic programming approach to the intelligent monitoring of anomalous human behaviour. In Paulus, D., Fuchs, C., Droege, D., eds.: OGRW2014, Koblenz, University of Koblenz-Landau (2015) 82–85
20. Fisher, R.: CAVIAR test case scenarios. The EC funded project IST 2001 37540. [Online] Available from: http://homepages.inf.ed.ac.uk/rbf/CAVIAR/ (2007)
21. Fisher, R.: BEHAVE: Computer-assisted prescreening of video streams for unusual activities. The EPSRC project GR/S98146. [Online] Available from: http://groups.inf.ed.ac.uk/vision/BEHAVEDATA/INTERACTIONS/ (2013)
22. Morozov, A.A.: A demo on linking Java3D with Actor Prolog. [Online] Available from: http://forum.jogamp.org/Demo-on-linking-Java3D-with-Actor-Prolog-tt4028018.html (2013)
23. Gothel, S.: The JOGL project Web Site. [Online] Available from: http://forum.jogamp.org (2015)
24. Morozov, A.A.: Visual logic programming based on the SADT diagrams. In Dahl, V., Niemela, I., eds.: ICLP 2007. LNCS 4670, Heidelberg, Springer (2007) 436–437