Incorporation of Distributed Multi-Agent Programming Means in a Strongly Typed Logic Language

Alexei A. Morozov, Olga S. Sushkova,
Alexander F. Polupanov

Kotel'nikov Institute of Radio Engineering and Electronics of RAS
Mokhovaya 11-7, Moscow, Russia
http: // www.fullvision.ru / actor_prolog
morozov@cplire.ru o.sushkova@mail.ru sashap55@mail.ru

An example of a declarative multi-agent system for intelligent visual surveillance

There are three agents implemented in the distributed logic language. The first agent acquires video. The second agent detects running people. The third agent detects abandoned things.

The plan of the report

- The agent approach to the intelligent visual surveillance / the real-time analysis of video.
- The declarative approach to the agent programming.
- The distributed version of the Actor Prolog object-oriented logic language.
- The problem of incorporation into the logic language the ability of remote procedure calls.
- A combined type system which provides a solution of the problem of the strong typing in the multi-agent systems.

The multi-agent approach to the intelligent visual surveillance

The idea is in that the intelligent visual surveillance system consists of communicating programs (agents) that have the following properties:

- Autonomy. Agents operate without direct control from users and other agents.
- Social ability. Agents can co-operate to solve the problem.
- Reactivity. Agents perceive the environment and respond to external events.
- Pro-activity. Agents demonstrate a goal-directed behavior.

Are the agents useful for the intelligent visual surveillance indeed?

Agents are used for different purposes in various research projects on the image processing / intelligent visual surveillance:

- Some researchers use agents just because of the fashion.
- Different agents control different hardware units; for instance, pan-tilt-zoom (PTZ) cameras.
- Different agents implement different functions / types of analysis. The example was in the first slide.
- Mobile agents are useful if it is preferably to transfer the executable code, but not the data in the network.
- Different agents analyze different areas of space, for instance, different rooms in a building.

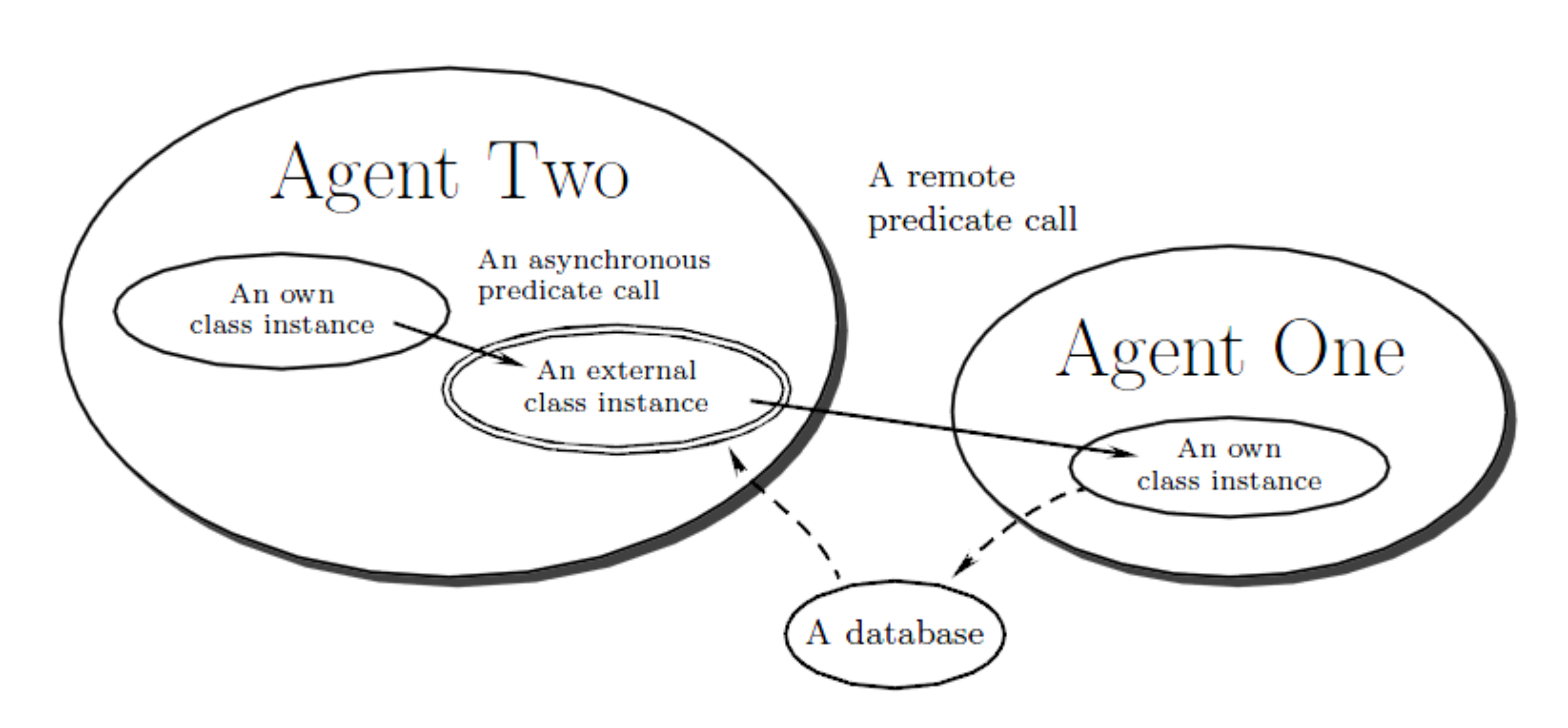
The declarative approach to the agent programming

- Prolog-like syntax for beliefs, rules, goals, plans, etc. is widely used in the area of intelligent agents.
- In the intelligent visual surveillance, the agents are to perform very specific operations on big arrays of binary data that are out of the framework of the conventional symbolic processing operations.
- The distinctive feature of our approach is in that we implement the intelligent video analysis using the concurrent object-oriented language Actor Prolog and a compiler of Actor Prolog into pure Java.
- The Actor Prolog language differs from other Prolog-based agent languages in that it is not based on the belief-desire-intention (BDI) model and it does not directly offer high-level agent features. Actor Prolog is rather a high-performance object-oriented logic language that is a base for implementation of real time multi-agent applications.

The distributed version of the Actor Prolog object-oriented logic language

- Previously, we have demonstrated that the translation of the object-oriented logic language into Java yields a sufficiently fast executable code for real time video analysis and detection of complex patterns of the abnormal people behavior.
- This approach can be extended to the distributed visual surveillance, because the Actor Prolog language is indeed an object-oriented language and can be easily adapted to the distributed programming framework even without modifications of the syntax.
- The only problem to be solved was the incorporation into the language the ability of remote procedure calls.

Incorporation into the logic language the ability of remote procedure calls



Agent One publishes an instance of a class in the external database. Then, Agent Two obtains this class instance and sends an asynchronous message to this class instance using Java RMI.

The problem of the strong typing in multi-agent systems

- There is a contradiction between the strong type system of the Actor Prolog language and the idea of the independency of the agents.
- The strong type system is necessary for generation of fast and reliable executable code.
- One needs to transfer information about the data types between the software units to implement their link and static type-checking.
- This kind of information exchange between the agents is undesirable, because it decreases the autonomy of the agents.
- The following solution of the problem is proposed: the type system of the Actor Prolog language is partially softened to allow a dynamic type-checking (instead of the static one) in some restricted cases linked with the inter-agent communications.

A combined type system supports strong typing in the multi-agent systems

- The **Actor Prolog** language has the **strong type system** that supports various kinds of **simple** and **composite** data items like numbers, structures, lists, etc. The **static type-checking** and standard features of a **nominative type system** are used.
- At the same time, the **dynamic type-checking** and elements of a **structural type system** are implemented for all the **external worlds**.

A strong type system in the distributed Actor Prolog

- The Actor Prolog language supports both types (domains) and classes/objects.
- A distinctive feature of the language is in that the object and the data item notions are clearly separated in the language.
- Actor Prolog supports the following simple data types: integer, real, symbol, and string. The structural matching is straightforward.
- There are three kinds of composite types in Actor Prolog, namely: structures, lists, and so-called underdetermined sets.
- The data structure in Actor Prolog can include class instances. To verify a remote predicate call one needs to check the name and the arity of the predicate, the flow pattern of the predicate, a structural compatibility of all the arguments.

Examples of type definitions in the Actor Prolog language

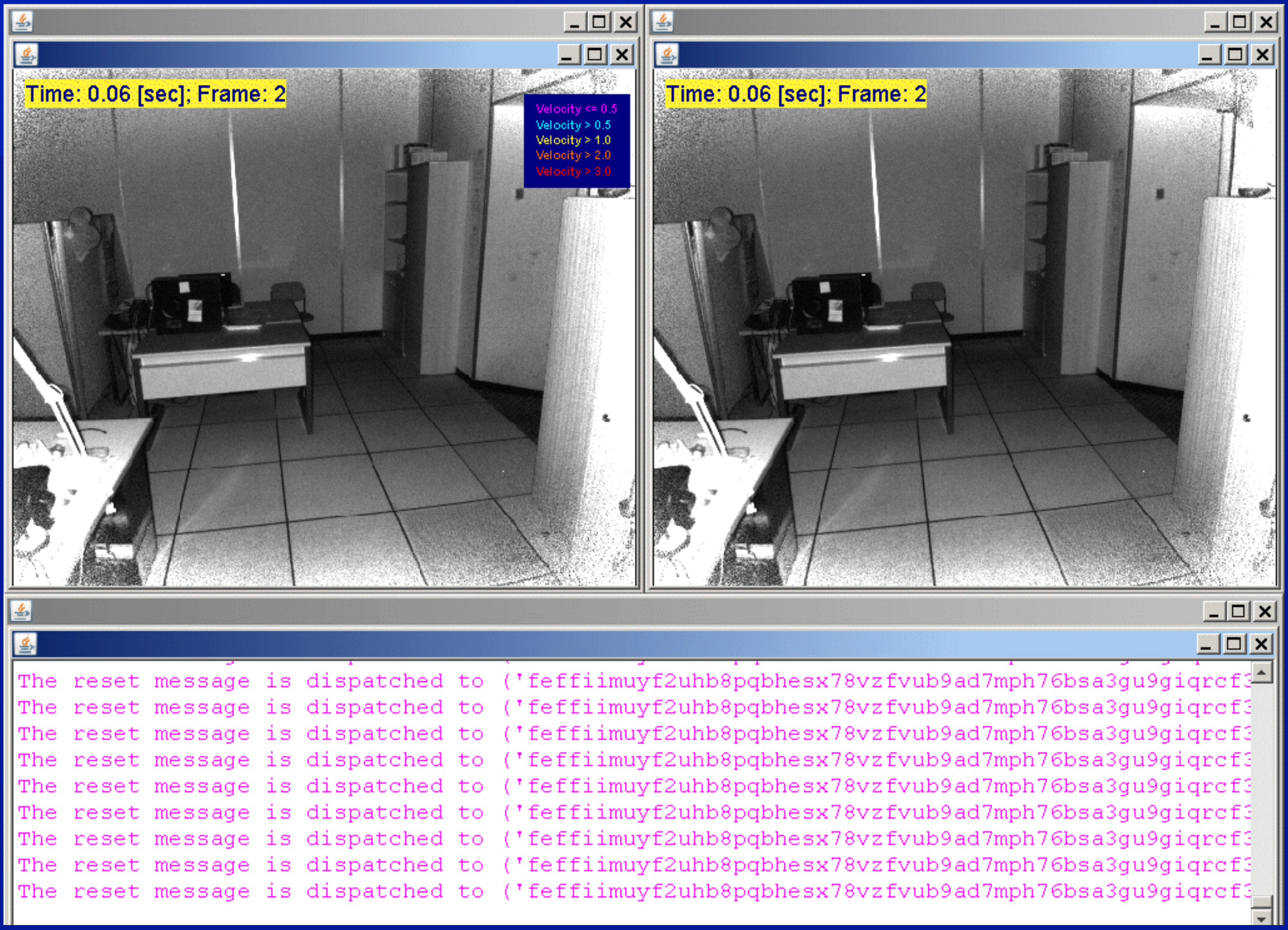
```
DOMAINS:
Year           = INTEGER.
Height         = REAL.
Color          = SYMBOL.
Message        = STRING.
Numerical      = INTEGER; REAL.
AppointedDate  = date(Year,Month,Day).
Dates          = AppointedDate*.
Customer       = {name: STRING, age: INTEGER}.
MessageHandler = ('MyClass').

PREDICATES:
intruder_coordinates(REAL,REAL)      - (i,i);
send_coordinates('AcceptingAgent')   - (i);
```

The structural matching of the external class instances

- The distributed Actor Prolog ensures that an instance of a class belongs to the class pointed in the type definition only if this class is defined in the same logic program.
- A check of an external class instance is to be performed only when the accepting program try to invoke a method in the external object.
- Thus, the implementation of this check requires information on the origin of all the objects in the logic program.
- Distributed Actor Prolog keeps internal tables of all the class instances transferred outside / accepted from other logic programs.

An example of a declarative multi-agent system for intelligent visual surveillance



The video data to be transferred between the agents is encapsulated in built-in class instances. It is not necessary to encode huge video files using Prolog terms. The data exchange is based on Java RMI.

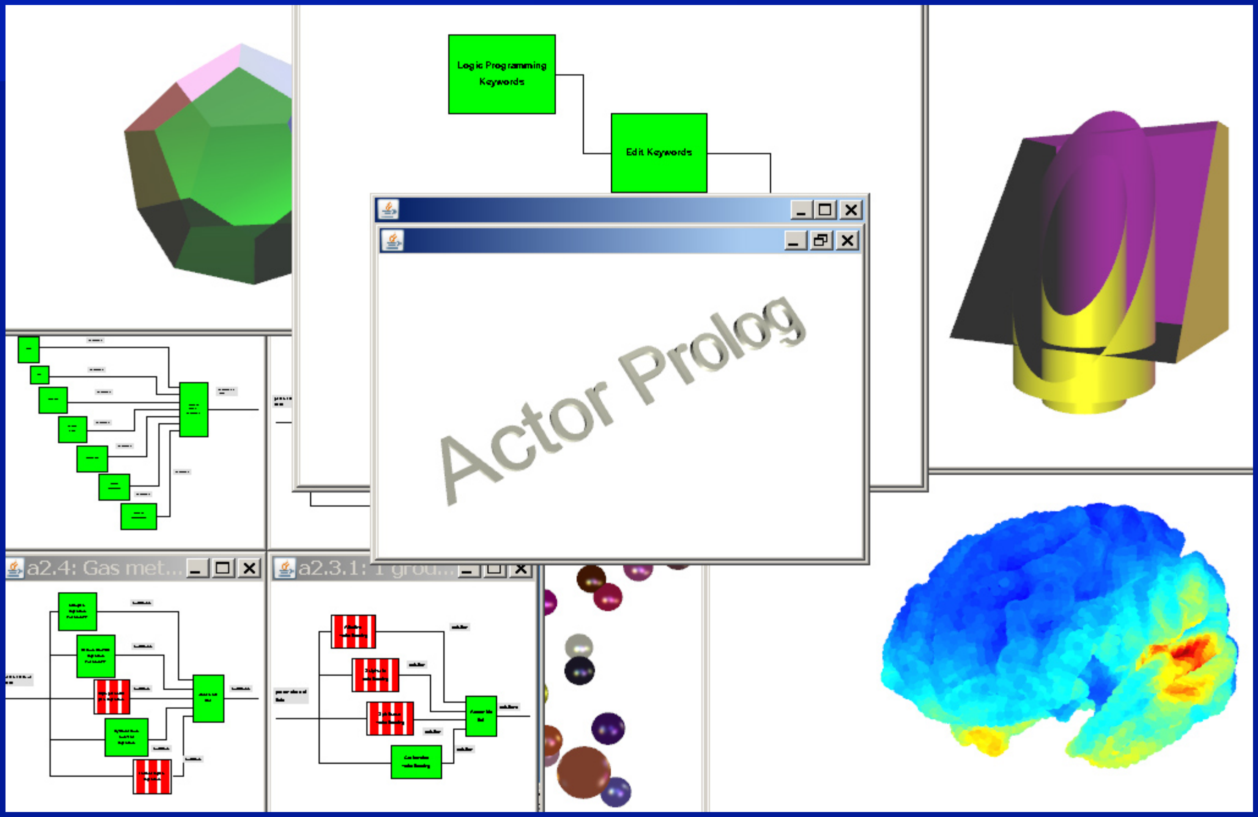
Conclusions

- The extension of the Actor Prolog language with the ability of distributed logic programming was developed.
- A combined type system is developed which ensures the advantages of the static type-checking for the generation of the fast executable code and the flexibility of the dynamic type-checking that is necessary for the multi-agent systems design.
- The distributed extension of the Actor Prolog language gives new means for experimenting with the real-time multi-agent logic programming and practical applications of the logic programming in the intelligent visual surveillance.

Some Web Resources

- The Project Web Site containing demo video clips, applets, and source code of Actor Prolog demo programs for intelligent video surveillance: http://www.fullvision.ru/actor_prolog/
- A GitHub repository containing source codes of Actor Prolog built-in classes: https://github.com/Morozov2012/actor-prolog-java-library
- Getting Started in Actor Prolog: http://www.cplire.ru/Lab144/start/
- A demo Web Site on linking Java3D with the Actor Prolog language: http://alexei-morozov-2012.narod.ru/

The Actor Prolog translator to Java is available



You are welcome to participate in the development, beta testing, and application of the Actor Prolog system.

Thank you

http://www.fullvision.ru/actor_prolog

What is Actor Prolog?

- **Actor Prolog is a logic language designed on the basis of our experience of business / industrial applications of logic programming.**
- **Actor Prolog was initially designed as an object-oriented language with a classical model-theoretic semantics.**
- **Actor Prolog is a concurrent language.**
- **Actor Prolog implements underdetermined sets.**
- **Actor Prolog supports domain and predicate declarations; that is very important for development of big / industrial programs.**
- **Actor Prolog produces a fast, stable, and portable stand-alone executable code (including Java applets).**
- **The Actor Prolog programming system is open; it can be easily extended by new built-in classes. For instance, Java2D and Java3D are connected with the Actor Prolog system in this way.**

Why Translation to Java?

- It was our unsuccessful attempt to use *a commercial Prolog* for programming Web agents several years ago. The programs crashed after several days of work because of unintelligible internal problems in the translator and libraries.
- We need a reliable implementation of a logic language with a clear memory management. Logic programs operating with big amounts of data should work stably during long periods of time.
- We can use an industrial Java virtual machine as a basis for a logic programming system, because 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 programs.
- Nevertheless, we need a fast executable code that is appropriate for real-time data processing.

The Compilation Schema

- Source text scanning and parsing. Methods of thinking translation that prevent unnecessary processing of already translated source files are implemented.
- 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.
- Type check. The translator checks data types of all predicate arguments and arguments of all class instance constructors.
- Determinism check. The translator checks whether predicates are deterministic or non-deterministic. So-called imperative predicates are supported, that is, the compiler can check whether a predicate is deterministic and never fails.
- A global flow analysis. The compiler tracks flow patterns of all predicates in all classes of the program.
- Generation of an intermediate Java code.
- Translation of this code by a standard Java compiler.

The Compilation Schema

- The imperative predicates are translated to Java procedures directly. The imperative predicates usually constitute the main part of the program and ensure very high level of code optimization.
- The 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.
- 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 Compilation Schema

- Tail recursion optimization is implemented for recursive predicates. Recursive predicates are implemented using the while Java command.
- The Actor Prolog language supports explicit definition of ground / non-ground domains and the translator uses this information for deep optimization of the executable code.
- The Actor Prolog language is significantly different from the conventional Clocksin & Mellish Prolog. Object-oriented features and supporting concurrent programming make translation of an Actor Prolog code to be a complex problem.

The Imperative Predicates

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

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!"));
}
```

The Deterministic Predicates

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

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

- Backtracking is implemented using a special kind of light-weight Java exceptions.
- Tail recursion optimization is possible.

The Non-Deterministic Predicates

```
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);
      }
}
```

- A standard method of continuation passing is used.
- Tail recursion optimization is possible.

An Imperative Predicate P calls a Non-Deterministic Predicate Q

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

```
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);
   }
}
```

```
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();
      }
   }
}
```

Extension of Actor Prolog

```
package "Morozov/Vision":
class 'ImageSubtractor' (specialized 'Alpha'):
  extract_blobs        = 'no';
  track_blobs          = 'no';
...
[
SOURCE:
  "morozov.built_in.ImageSubtractor";
CLAUSES:
subtract(FrameNumber,Image):
  [external "subtract"].
...
]

public static abstract class
  AbstrCls5_1076_ImageSubtractor extends
  morozov.built_in.ImageSubtractor {
...
```

Actor Prolog Benchmark Testing

(Intel Core i5-2410M, 2.30 GHz, Win7, 64-bit)

Test	Iter. No*	Actor Prolog to Java 64-bit	SWI-Prolog v. 7.2.2
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	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

*Benchmark time is measured in milliseconds per iteration.

We invite you to participate in the beta testing of the educational version of Actor Prolog:

- Domain and predicate declarations are supported.
- One can switch off the check of the declarations.
- Creation of And-Or trees is supported.
- Compilers to Java and EXE code are available.

morozov@cplire.ru

http: // www.fullvision.ru / actor_prolog