Logic Programs with Access Modifiers. Adding Events for Query Management

Adrian Giurca

Institute of Informatics

Brandenburg University of Technology at Cottbus,

GERMANY

Abstract : The aim of this paper is to propose a framework for logic programs cooperation using ALP model started in [6]. Following [12], the paper analyzes the communication events TELL, ASK-IF, REPLY-IF, ASK-ALL, REPLY-ALL needed in interactive query answering. The main section (Section 3) presents the MOF/UML model of our framework and the ALP behavior when a communication event (ALP `change_state` operation) is received. In Section 4 we provide an example of cooperation between two (or more) logic programs under minimal model semantics using ALP programs and event communication.

Keywords: distributed environments, logic programming, rules, communication events

Adrian Giurca obtained his PhD in 2004 from University of Bucharest, Romania, in Mathematics and Computer Science. During the period 2000-2004 he was a lecturer with the Faculty of Mathematics and Computer Science from University of Craiova, Romania. Presently he is a researcher in the Department of Internet Technology in the Institute of Informatics from Brandenburg University of Technology at Cottbus, Germany. His main research fields are: Uncertainty Representation, Semantic Web, Logic Programming and Multi-Agent Systems.

1. Preliminaries

This paper follows our previous research from [6] where we proposed a class of logic programs called logic programs with access modifiers (ALP) and a global cooperative knowledge architecture (ALP framework) based on this class of programs. Let A be an ALP. We denote PRED*(*A*)* the set of all predicate symbols from A.

1. Following the principle of dominant queries the designer declares the set of public predicates from A, denoted by PRED(A)public. Usually, the public part of a logic program is a set of clauses which is constructed following the most queried predicates from the program. This part is necessary to be public because a participant it in order to resolve his goals. The other predicates are used by the program himself to solve the participant goal. See also [6] Definition 2, for details.
2. The set of all protected predicates denoted PRED(A)protected. The protected part of a program which is a set of clauses concerning predicates that must use the public part of the program A, in order to solve queries ([6] Definition 2).
3. The set of all private predicates from A, denoted PRED(A)private1. The corresponding set of private clauses is used only in internal reasoning of A.

In [6] Proposition 1 we prove that, having declared $PRED(A)_{public}$ we can construct all the rest of sets from an ALP. An architectural design of an ALP is depicted in Figure 1.

1 Like in [6], $PRED(A)_{\text{private}} = PRED(A) - (PRED(A)_{\text{public}} \cup PRED(A)_{\text{protected}})$

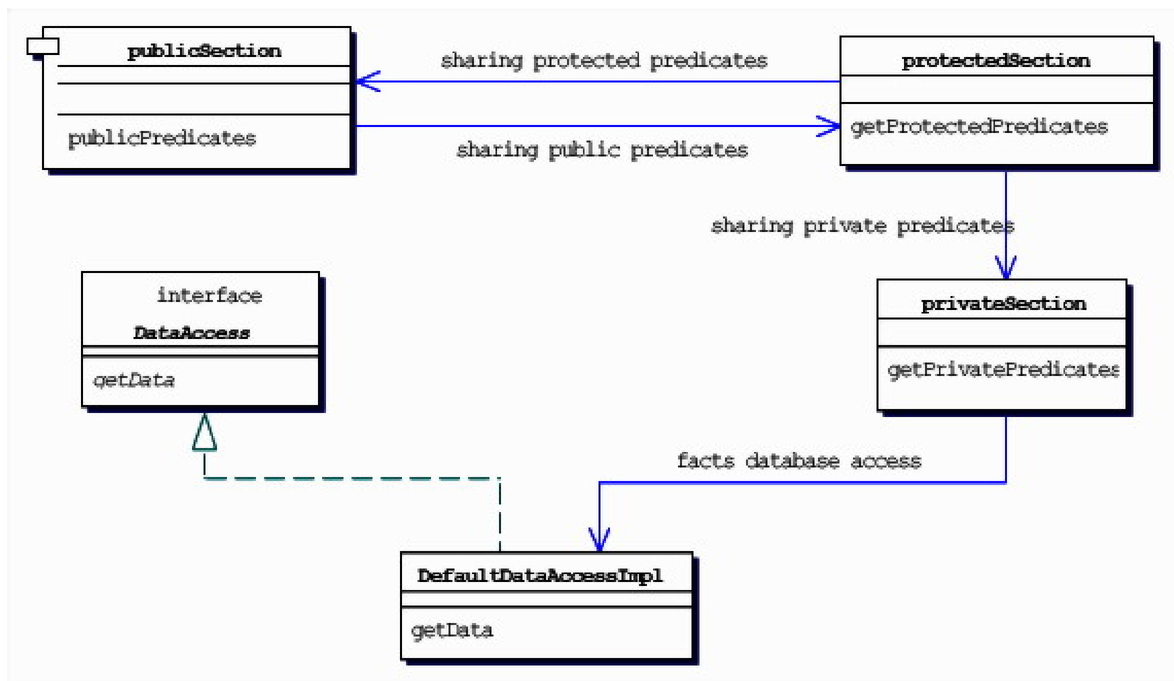

Figure 1: ALP Structure, [6].

In this diagram there exists a separation between public, protected and private predicates to restrict participant access only to public predicates. Also, we have a concrete separation between the layer of private clauses and facts, which are obtained from a database. This corresponds of deductive databases point of view. At that time we proposed also a framework for using ALP's in a distributed environment. The architecture of this framework is depicted in Figure 2. The main components are:

1. Users. Can be any human users or artificial agents that can understand the communication channel.
2. ALP. Logic programs with access modifiers (ALP's). These logic programs are fragmented using access modifiers using the principle of dominant queries.
3. KM. Knowledge Manager.

The main disadvantage of this design is a too strong role assigned to the KM. In fact the responsibility of the communication acts is assigned to the KM so, the KM must manage almost all the conversational acts in the system. In order to obtain a mode flexible architecture and to enable "free" knowledge sharing between entities from our framework, we investigate the possibility for the treatment of queries using communications acts represented as events.

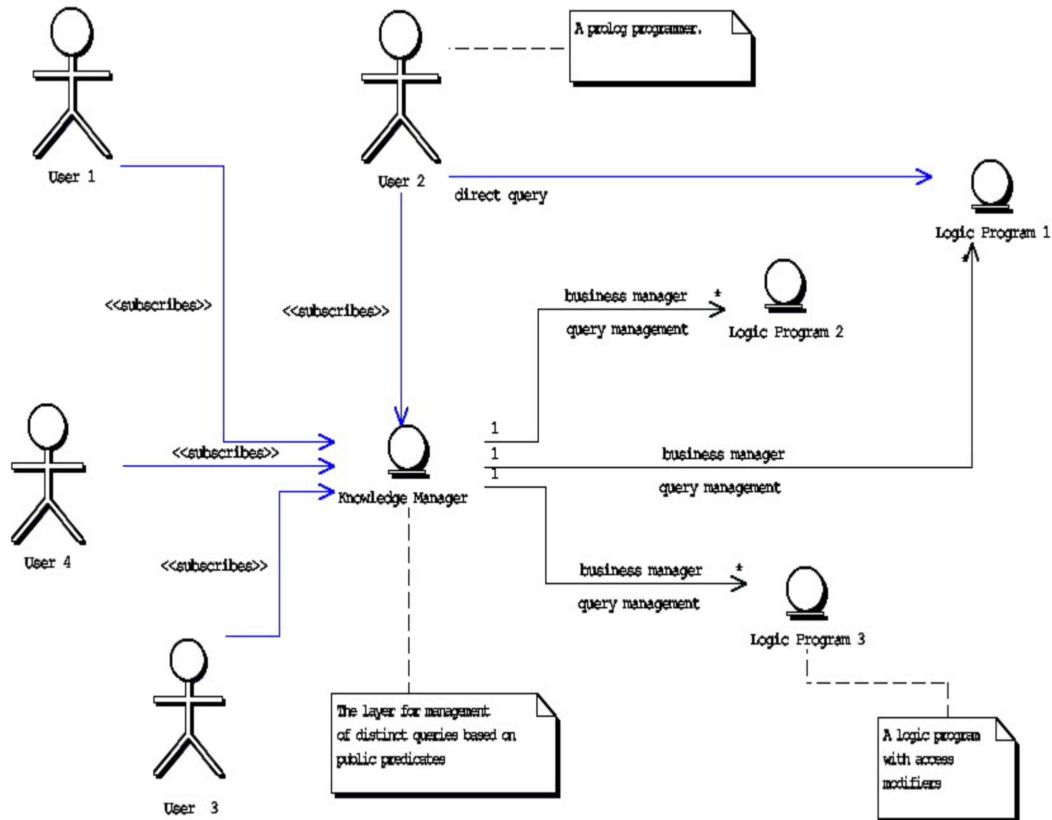

Figure 2: Global Knowledge Architecture [6].

2. Five Kind of Communication Events

The set of possible communication acts includes telling, asking, replying, and more. This paper will analyze only these three kinds of events. Following [12], message types for expressing those communication events needed in interactive query answering include TELL, ASK-IF, REPLY-IF, ASK-ALL, REPLY-ALL. Notice that communicative acts can be directly performed, can be planned and can be requested by any participant.

TELL. We use TELL for supplying new information to the system. For example, asserting a new knowledge into an ALP database corresponds to sending a TELL*(*equity*(*"RRC","Rompetrol","ROTX"*))* message with the atomic sentence equity*(*"RRC", "Rompetrol", "ROTX"*)* as content. A TELL event can be received by any ALP. Note that, in this paper, the message content parameter of TELL is a ground atom.

ASK-ALL. Similarly, ASK-ALL*(*equity*(*X,Y,Z*))* corresponds to the first order query

$\forall$X, Y, Z equity*(*X, Y, Z*)*

The event delivers the collection of all answer substitutions in the form of a list.

ASK-IF. Using ASK-IF *(*equity*(*"RRC", ,Z*))* in our system we look in the system for an answer that can be a variable substitution set `S`, or the answer `"no"`. If there exists a substitution for variable Z in the system according to the public equity predicate then the answer will be the appropriate substitution, otherwise will be no or unknown (see also REPLY-IF). The message content of ASK-IF is an if-query corresponding to the first order formula

$\exists$Z equity *(*"RRC", _, Z*)*

REPLY-IF. This kind of event can be fired by an ALP (or any other participant) as a response to an ASK-IF received event. A REPLY-IF event has two parameters: the if-query from the received ASK-IF event and the answer of the system. There are three possible cases:

- If we have equity(”RRC”, ”Rompetrol”, ”ROTX”) in the database of the ALP_1 and equity is a public predicate and ALP_1 receive an ASK-IF event like ASK-IF(equity(”RRC”, ,Z)) it can fire the event

  REPLY-IF (equity (”RRC”, _ , Z), (Z=”ROTX”)).

- If we don’t have equity(”RRC”, ”Rompetrol”, ”ROTX”) in the database of the ALP_1 and equity is a public predicate, then ALP_1 can fire an event like

  REPLY-IF (equity (”RRC”,_ ,Z), no).

- If equity is not a public predicate, then ALP_1 can fire an event like

  REPLY-IF (equity (”RRC”,_ , Z) ,unknown).

REPLY-ALL. Similarly, the REPLY-ALL event can be fired by system participants (ALP’s or other participants) as a reply to an ASK-ALL event. In the same manner it has two parameters: the query and the response which is the list of all corresponding instances or empty list if we not have any appropriate instance. So, it is possible to obtain an event like

REPLY-IF (equity(”RRC”, _ ,Z),[(Z=”ROTX”), (Z=”BET”), (Z=”BET-C”)])

3. Using Communication Events to Modeling Queries

This section is devoted to query modeling using communicative events. In order to realize this task we must follow two steps: first, we develop the basic constructs of the query language then we annotate the ALP’s with these constructs.

The main thing is that every ALP can have his personal query constructs. In order to obtain a sound model it is required that the principle of “non-fail events” must hold. (see also [7] and [3]).

That is, every fired event is accepted by some receiver who must send a response to some participant from the system.

- TELL indicates that the sending participant:
  - Holds that some proposition is true,
  - Intends that the receiving participant also comes to believe that the information is true (the sending agent is sincere, and has (somehow) generated the intention that the receiver should know the proposition (perhaps it has been asked)),
  - Does not already believe that the receiver has any knowledge of the truth of the information.

  From the receiver’s viewpoint, receiving a TELL event entitles it to believe that:
  - The sender believes the information that is the content of the message, and
  - The sender wishes the receiver to believe that information also.
- **The informal meaning of** ASK is *“asking a participant about the truth of specific information”*. A participant can perform an ASK event if:
  - Has no knowledge of the truth value of the information, and,
  - Believes that the other participant can inform the sender if it knows the truth of the information.

We denote the set of all communicative events by `CommEvents`2 and the set of all ALP’s with **ALP**.
The basic meta-predicates of our event rules are `sendMsg` and `recvMsg`.

Definition 1 (Event Pattern) The general communication event pattern is a simple structure of the form: `event-name(atom[,answer])`

2 CommEvents = {TELL, ASK−IF, ASK−ALL, REPLY−IF, REPLY−ALL}

where atom is a first order atomic formula and answer is one of the following: a substitution, a set of substitutions, NO or UNKNOWN.

The Change State of an ALP receiving Communication Events

Below we model the ALP behavior when receive a communication event. As we already say the communication events can occur in our framework are TELL, ASK-IF, REPLY-IF, ASK-ALL and REPLY-ALL. All algorithms contain the following notations and usual functions used in list (set) management:

A,B are ALP's;

p(t1,...tn) is an atom;

pred_A - the set of all predicates from A;

public_A - the set of all declared public predicates from A;

protected_A - the set of all (obtained) protected predicates;

private_A - the set of all (obtained) private predicates;

assert(atom) - the common assert function used to update the facts database of an ALP;

remove(param) - the usual removal from a list (set). If the param is not in the list (set) the function call has no effect;

add(param)- the usual add of an element into a list (set). If the param is already in the list (set) the function call has no effect;

isEmpty() - the usual empty set test;

calculate(public(A)) - the function which obtain the fragmentation of A (from [2], Proposition 1);

compute(atom) - the usual atomic query computation in a logic program;

State change when receiving a TELL event

```
//The test can be changed. This version is only for illustrative
//purposes. It does not have need to be optimal.
change-state(recvMsg(TELL(p(t1,...,tn)),B)){
 if ( protected_A.contains(p) || private_A.contains(p)) {
  // declare p public in A
   private_A.remove(p);
   protected_A.remove(p);
   public_A.add(p);
   A.calculate(public_A);
  } else {
   if (!public_A.contains(A)){
   public_A.add(p);
   }
 }
 // update facts database in A
 A.assert(p(t1,...,tn));
}
```

State change when receiving a REPLY-IF event

```
change-state(recvMsg(REPLY-IF(p(t1,...,tn),answer),B){
 if ( !(answer == NO || answer == UNKNOWN)) {
   // update facts database in A
   A.assert(p(answer(t1),...,answer(tn)));
   if (p(answer(t1),...,answer(tn)) is ground){
   sendMsg(TELL(p(answer(t1),...,answer(tn))),KM);
  }
 }
}
```

State change when receiving an ASK-IF event

```
change-state(recvMsg(ASK-IF(p(t1,...,tn)),B)){
 if ( public_A.contains(p)) {
   if ( exists sigma such that A entail
             p(sigma(t1),...,sigma(tn))){
     A.sendMsg(REPLY-IF(p(t1,...,tn),{sigma}),B);
    } else{
       //q is the first atom that fails in the resolution proof
       A.sendMsg(REPLY-IF(p(t1,...,tn),NO(q)),B);
       A.sendMsg(ASK-IF(q),KM);
    }
  } else {
     A.sendMsg(REPLY-IF(p(t1,...,tn),UNKNOWN),B);
     A.sendMsg(ASK-IF(p(t1,...,tn)),KM);
  }
}
```

State change when receiving a REPLY-ALL event

```
change-state(recvMsg(REPLY-ALL(p(t1,...,tn),answer),B){
 if ( !(answer == NO || answer == UNKNOWN)) {
  // update facts database in A
  for all ansver_i in answer {
   A.assert(p(answer_i(t1),...,answer_i(tn)));
   if (p(answer_i(t1),...,answer_i(tn)) is ground){
     sendMsg(TELL(p(answer_i(t1),...,answer_i(tn))),KM);
   }
  }
 }
}
```

State change when receiving an *ASK-ALL* event

```
change-state(recvMsg(ASK-ALL(p(t1,...,tn)),B)){
 if ( public_A.contains(p)) {
  //compute the all substitutions sigma_i such that
  //A entail p(sigma_i(t1),...,sigma_i(tn))
  S = A.compute(p(t1,...,tn));
  if ( S.isEmpty()) {
   //q is the first atom that fails in the resolution proof
   A.sendMsg(REPLY-IF(p(t1,...,tn),NO(q)),B);
   A.sendMsg(ASK-ALL(q),KM);
  }else{
   A.sendMsg(REPLY-IF(p(t1,...,tn),S),B);
  }
  }else {
   A.sendMsg(REPLY-ALL(p(t1,...,tn),UNKNOWN),B);
   A.sendMsg(ASK-ALL(p(t1,...,tn)),KM);
  }
}
```

The MOF/UML Model of the framework

In the Figure 3 we see the MOF/UML model of our framework. Figures 4, 5 and 6 presents the sequence diagrams related to the corresponding event receiving.

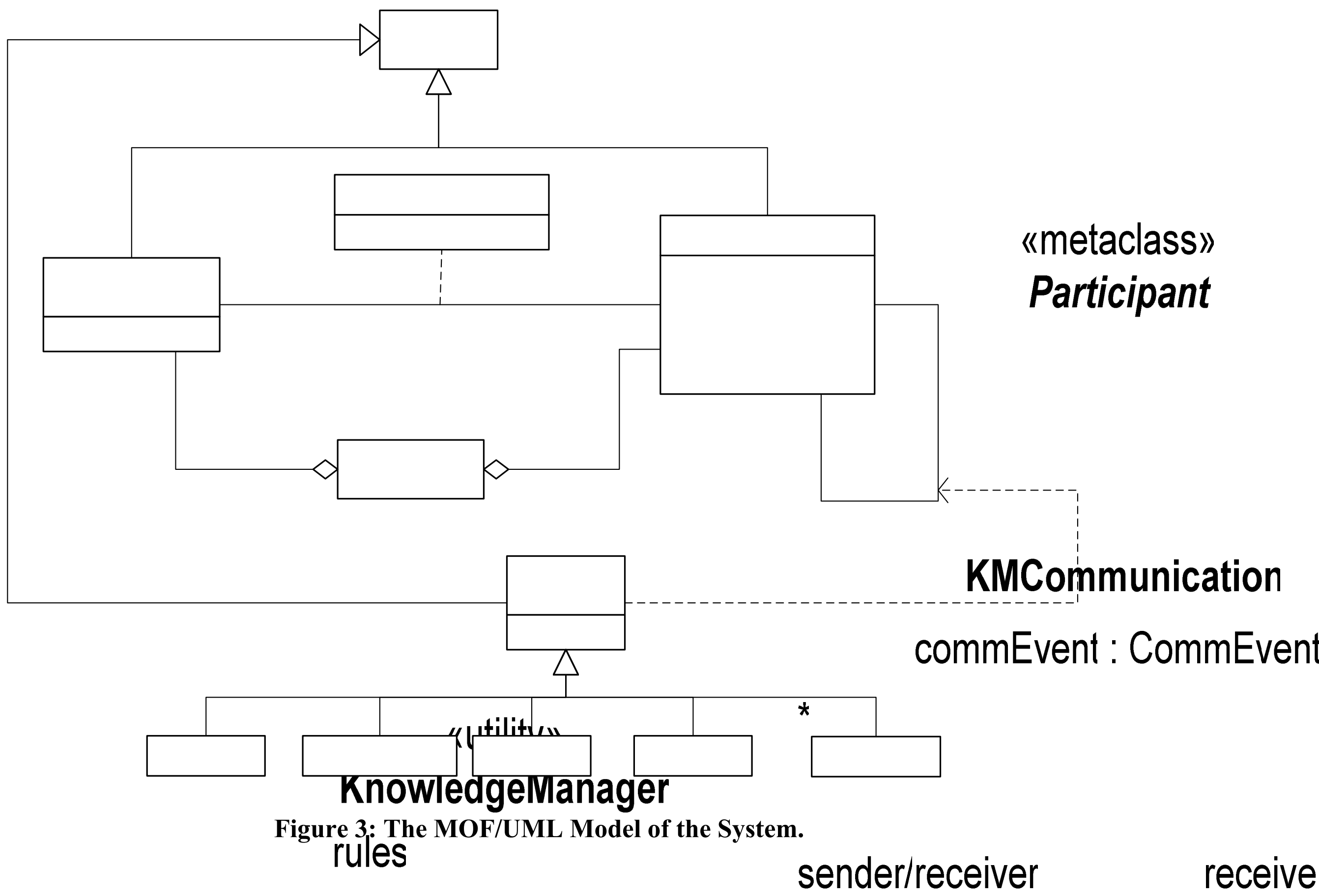

Figure 3: The MOF/UML Model of the System.

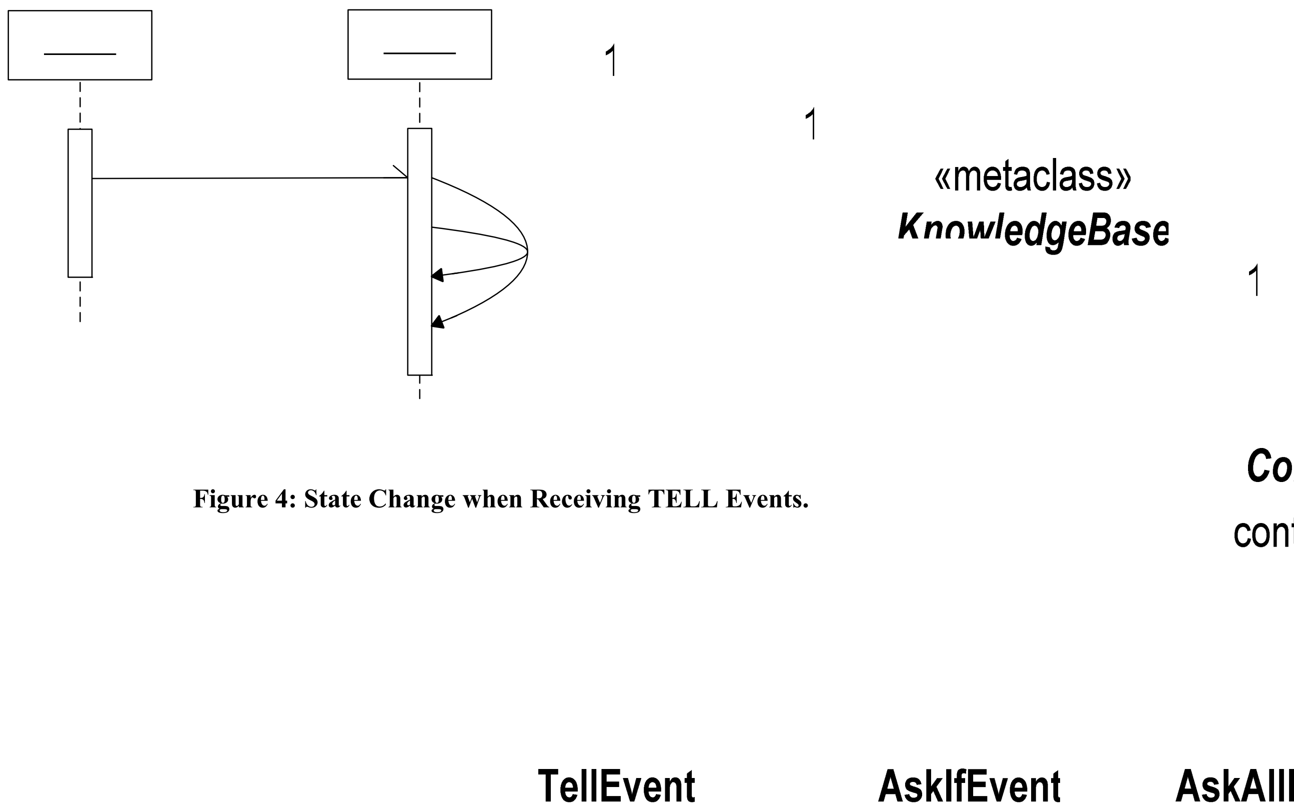

Figure 4: State Change when Receiving TELL Events.

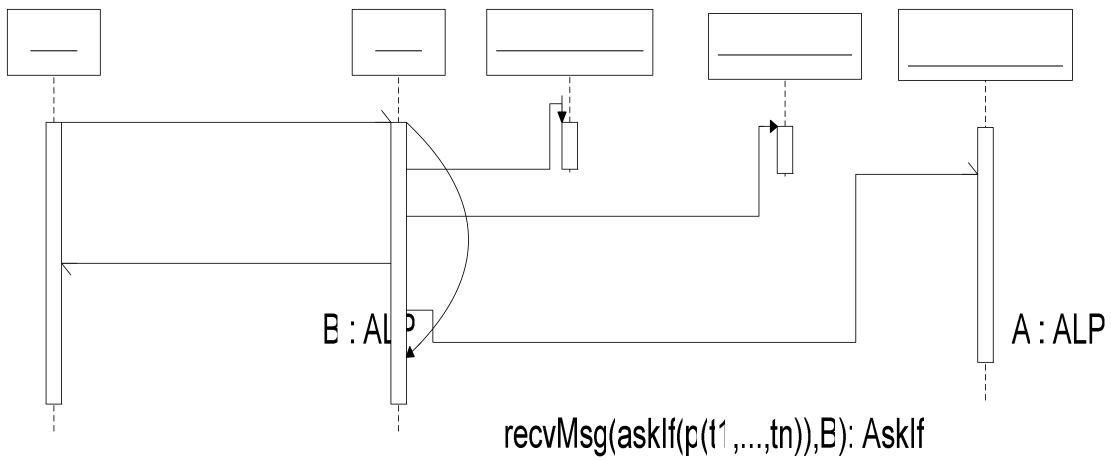

Figure 5: State Change when Receiving ASK-IF or ASK-ALL Events

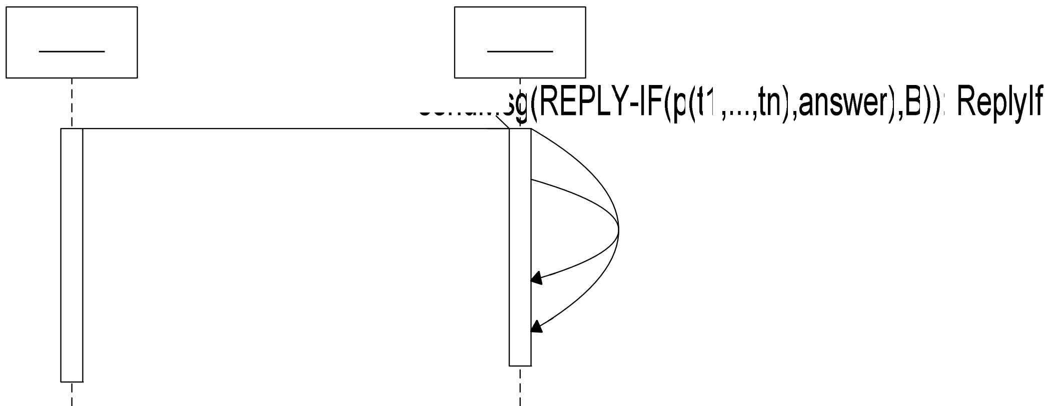

Figure 6: State Change when Receiving REPLY-IF or REPLY-ALL Events

Communication Rule Patterns in KM

The role of the Knowledge Manager (KM) is restricted now only to manage communication events that cannot be solved by one of the participants in the system. In order to be able to do this KM must provide rules for managing communication events. In our construction the following kind of rules can be modeled:

Definition 2 (Rules Pattern) The following patterns are event rules or communication rules:

1. **Inform Rule (IR)**:

$$sendMsg(e, P) \leftarrow$$

2. **Composite Inform Rule (CIR)**:

$$sendMsg(e, P) \leftarrow recvMsg(e_1, P_1) \wedge \ldots \wedge recvMsg(e_n, P_m).$$

3. **Request Rule (RR)**:

$$recvMsg(e, P) \leftarrow$$

where $e, e_i \in CommEvents$ and $P, P_j \in ALP$, for all $i=1,\ldots,n$ and $j=1,\ldots,m$.

4. Case Study: Logic Programs Cooperation under Minimal Model Semantics

In this section we provide an example of cooperation between two (or more) logic programs under minimal model semantics.

Minimal Model Semantics is not Decomposable

In our paper [5] we provide some examples proving that, unfortunately, the Herbrand semantic of Horn logic programs can't be obtained by composition from the semantic of program clauses itself (actually, the least Herbrand model of a program cannot be obtained, in general, from the least Herbrand model of it's clauses).

We also, use a counterexample in order to prove that it is not possible to make set operations with clauses from P_1 and P_2 for obtaining a program $P_1 \oplus P_2$ such that

$M_{P_1 \oplus P_2} = M_{P_1} \cup M_{P_2}$

and a program $P \subseteq P_1 \cup P_2$ such that

$M_P = M_{P_1 \cup P_2} - (M_{P_1} \cup M_{P_2}) = \{p(v), r(u)\}$

where M_P denote the least Herbrand model of P.

For example, if we want to construct P by collecting the clauses involved in the deduction of $\{p(v)\}$, then we must take the clauses $p(X) \leftarrow q(X)$ and $q(v) \leftarrow$ but, in this case, $p(u) \in M_P$.

Example 1 (See [5]) Let $P_1, P_2 \in \mathbf{P}$ such that

$$P_1 = \begin{cases} p(X) \leftarrow q(X) \\ q(u) \leftarrow \end{cases} \text{ and } P_2 = \begin{cases} r(X) \leftarrow q(X) \\ q(v) \leftarrow \end{cases}$$

Then $M_{P_1} = \{q(u), p(u)\}$ and $M_{P_2} = \{r(v), q(v)\}$. Clearly, $M_{P_1} \cup M_{P_2} = \{q(u), p(u), r(v), q(v)\}$. Now, if we consider the union of clauses from P_1 and P_2, we obtain the program

$$P_1 \cup P_2 = \begin{cases} p(X) \leftarrow q(X) \\ r(X) \leftarrow q(X) \\ q(u) \leftarrow \\ q(v) \leftarrow \end{cases}$$

and $M_{P_1 \cup P_2} = \{q(u), p(u), r(v), q(v), p(v), r(u)\}$.

A Scenario: Using ALP with Events to Model a Cooperative Architecture

In this section we prove that using the ALP with events framework we can obtain some desired results in cooperation between logic programs.

Scenario 1. Let $P_1, P_2 \in$ P as in Example 1.

We declare `PRED(P1)`$_{public}$ `= {p}` and `PRED(P2)`$_{public}$ `= {r, q}`.

Following the algorithm described in [6] we obtain:

$PRED(P1)_{protected} = \varnothing$,

$PRED(P1)_{private} = \{q\}$,

$PRED(P2)_{protected} = \varnothing$ and

$PRED(P1)_{private} = \varnothing$.

The following is a scenario of cooperation:

```
SYSTEM at TIME=t1
=================
1. Participant: sendMsg(ASK-IF(p(v)),P1)
2. P1: recvMsg(ASK-IF(p(v)),user) so,
            call: change_state(recvMsg(ASK-IF(p(v)),user))
      2.1 sendMsg(REPLY-IF(p(v),NO(q(v))),user);
      2.2 sendMsg(ASK-IF(q(v)),KM)
3. KM: recvMsg(ASK-IF(q(v)),KM) so, sendMsg(ASK-IF(q(v)),P2)
// KM send query to other participants
4. P2: recvMsg(ASK-IF(q(v)),KM) so
            call:change_state(recvMsg(ASK-IF(q(v)),KM))
      4.1 sendMsg(REPLY-IF(q(v),[]),KM) // q(v) is true in P2
5. KM: recvMsg(REPLY-IF(q(v),[]),P2) so,
      sendMsg(TELL(q(v)),P1) // tell all the new knowledge
6. P1: recvMsg(TELL(q(v)),KM) so call:change_state(TELL(q(v)),KM)
      6.1 P1 declare q as a public predicate
      6.2 P1 recalculate his public, protected and private predicates
      6.3 P1 add q(v) in his facts database

SYSTEM at TIME=t2
=================
1. Participant: sendMsg(ASK-IF(p(v)),P1)
2. P1: recvMsg(ASK-IF(p(v)),user) so,
            call: change_state(recvMsg(ASK-IF(p(v)),user))
      2.1 sendMsg(REPLY-IF(p(v),[],user); // p(v) is true in P1
....
```

As a conclusion, if at the first attempt the program P_1 fails to the query `p(v)`, at the second attempt it obtain the answer true using cooperation started in the first attempt.

5. Related Work and Future Research

Schroder et. al. [8], construct an argumentation language to specify argumentation protocols for multiagent systems and implemented multi-agent inference by vivid agents (for a complete information about vivid agents see Wagner [11]). Also in [9], they use PVM-Prolog to implement Vivid Agents.

Fischer at al. in [4] provides a large discussion about methodologies and principles of design agent based applications. They conclude that because there is no generally accepted and well-defined taxonomy for classifying agent-based applications.

Shoham in [10], provide a pioneering paper concerning the modeling behavioral systems as agents (both human and artificial).

Wagner in [11], provides the basis for vivid systems i.e. every knowledge system should be upwards compatible and conservatively extend, the system of relational databases.

Alferes et. al in [1] propose a foundational framework of evolving logic programs used to express and reason about dynamic knowledge bases. Also in [2] is proposed LUPS, a language for dynamic updates. They describe how different environment-aware behaviors of agents can be naturally expressed in LUPS. Also, the LUPS specification provides a formal declarative characterization of such behaviors, that can be exploited for performing resource-bounded analyzes and verifications.

As a future research, our plan is first to make an implementation model of the architecture, which will be useful to prove if such a system could be used in rules cooperation on the web and secondly, if the first step is successful, a formal semantics for this framework will be developed.

REFERENCES

1. ALFERES J.J., BROGI, A., LEITE, J.A. & PEREIRA, L.M., **Evolving logic programs**, In S. Flesca, S. Greco, N. Leone, and G. Ianni (editors) Logics in Artificial Intelligence, 8th European Conference on Logics in Artificial Intelligence (JELIA 02), Lecture Notes in Computer Science 2424, pages 50-62. 2002.

2. ALFERES J.J., BROGI, A., LEITE, J.A. & PEREIRA, L.M.,, **Computing environment-aware agent behaviours with logic program updates**, In A. Pettorossi (editor), ”Logic-Based Program Synthesis and Transformation - 11th International Workshop (LOPSTR’01), Selected Papers”, Lecture Notes in Computer Science 2372, pages 216-232, 2002.

3. Cohen P. R. and Levesque H. J., **Intention is Choice with Commitment**, In Artificial Intelligence, 42(2-3), pages 213-262, 1990.

4. FISCHER M., MOLLER J., SCHROEDER M., STANIFORD G., & WAGNER G., **Methodological Foundations of Agent-based Systems**, Knowledge Engineering Review, 12(3):323–329, 1997.

5. GIURCA A. & SAVULEA D., **An Algebra of Logic programs with Applications in Distributed Environments**, Annales of Craiova University, Mathematics and Computer Science Series, XXVIII, 2001, pp.147-159.

6. GIURCA A. & SAVULEA D., **Logic Programs with Access Modifiers**, 4th International Conference on Artificial Intelligence and Digital Communication, AIDC’2004, June 2004, pp. 22-31.

7. SEARLE J.R., **Speech Acts**, Cambridge University Press, 1969.

8. SCHROEDER M., MORA I. & ALFERES J. J., **Vivid Agents Arguing about Distributed Extended Logic Programs**.

9. SCHROEDER M. , MARQUES R., WAGNER G. AND CUNHA J., **CAP - Concurrent Action and Planning: Using PVM-Prolog to implement vivid agents**, In Proceedings of the 5th Conference on Practical Applications of Prolog, London, UK, April 1997.

10. SHOHAM Y., **Agent-oriented programming**, Artificial Intelligence, 60(1):5192, 1993.

11. WAGNER G., **A logical and operational model of scalable knowledge-and perception-based agents**, In Proceedings of MAAMAW96, LNAI 1038. Springer-Verlag, 1996.

12. WAGNER G., **Foundations of Knowledge Systems with Applications to Databases and Agents**, Kluwer Academic Publishers, 1998.