# Agent Mobility with Weak Local Inheritance and Transactional Remote Logic Invocation

Paul Tarau[1] and Arun Majumdar[2]

[1] Department of Computer Science and Engineering
University of North Texas
Denton, Texas, USA
*tarau@cs.unt.edu*
[2] Vivomind Research LLC.
Rockville, Maryland, USA
*arun@vivomind.com*

**Abstract.** We introduce a simple agent construct associated to a named local database and a "Twitter-style" weak inheritance mechanism between local agents.
On top of a remote predicate call layer, connecting distributed agent spaces, we build a replication mechanism allowing agents "visiting" remote spaces to expose their computational capabilities to non-local followers.
The resulting protocol has the remarkable property that only updates to the state of the agents are sent over the network through transactional remote predicate calls guaranteed to always terminate, and therefore spawning of multiple threads can be avoided. At the same time, calls to a visiting agent's code by its followers are always locally executed, resulting in performance gains and reduced communication efforts.
An implementation of the concepts discussed in this paper is available from `http://www.vivomind.com/lprolog/`.
**Keywords:** *agent oriented logic programming, agent mobility protocols, remote execution of logic programs, distributed dynamic Prolog databases, Java-based Prolog system*

## 1 Introduction

LeanProlog is a Java-based reimplementation of BinProlog's virtual machine, the BinWAM. It succeeds our *Jinni Prolog* implementation that has been used in various applications [1–4] as an *intelligent agent infrastructure*, by taking advantage of Prolog's knowledge processing capabilities in combination with a simple and easily extensible runtime kernel supporting a flexible reflection mechanism.

Agent programming constructs have influenced a significant number of mainstream software components ranging from interactive Web services to mixed initiative computer human interaction. From the very beginning, *Performatives* in Agent communication languages [5, 6] have made these constructs reflect explicitly the intentionality, as well as the negotiation process involved in agent interactions. At the same time, it has been a long tradition of logic programming languages [7–9], to use multiple logic engines for supporting concurrent execution and autonomous reasoning threads.

Naturally, this has suggested to investigate whether some basic agent-oriented language design ideas can be used for a refactoring of Prolog's interoperation with the external world, including interaction with other instances of the Prolog processor itself.

In this context we have centered our LeanProlog implementation around logic engine constructs providing an API that supports reentrant instances of the language processor. This has naturally led to a view of logic engines as instances of a generalized family of iterators called *Fluents* [9], that have allowed the separation of the first-class language interpreters from the multi-threading mechanism, while providing a very concise source-level reconstruction of Prolog's built-ins.

Later we have extended the original *Fluents* with a few new operations [10] supporting bi-directional, mixed-initiative exchanges between engines, high level multithreading constructs [11] and a coordination framework [12] resulting in an agent-oriented view of autonomous logic processors.

To ensure genuine scalability and readiness for integration in cloud-computing services, the next development stage involves designing resource-aware distributed agent programming constructs that are as oblivious as possible to the actual location of an agent. This paper describes a simple and efficient multi-agent layer that supports distributed execution, essential agent mobility and a simple and scalable multi-agent collaboration mechanism in LeanProlog.

The paper is organized as follows. Section 2 describes local inheritance mechanisms and interactions between agents. Section 3 describes our "remote logic invocation" (RLI) mechanism and its implementation in Java. Section 4 introduces agent spaces and section 5 explains the distributed programming constructs ensuring agent mobility. Section 6 gives some typical use cases of the framework. Finally, section 7 discusses related work and section 8 concludes the paper.

## 2 The Lightweight Prolog Agent Layer

We start with a quick "bird's eye view" of LeanProlog's multi-agent layer.

Our *agents* are implemented as *named* Prolog dynamic databases. Each agent has a process where its *home* is located - called an *agent space*. They share code using a simple "Twitter-style" mechanism that allows their *followers* to access to their predicates. The *A follows B* relation is not transitive, but agents can dynamically decide who they follow at a given time. Agents can also *visit* other spaces located on local or remote machines - where other agents might decide to follow their replicated "avatars". The state of their avatars is dynamically updated when a state change occurs in an agent's code space. Communication between agents, including avatar updates, is supported by a remote predicate call mechanism between agent spaces, designed in a way that each call is atomic and guaranteed to terminate.

We will next expand the details of various components.

## 2.1 Agents as Dynamic code in Multiple Named Databases

Lean Prolog provides dynamic database operations acting on multiple named databases. The name of the database is added as the first argument to otherwise usual Prolog predicates e.g. `assert(Clause)` becomes `db_assert(Database,Clause)`, `db_retract(ClauseHead)` becomes `db_retract(Database,ClauseHead)` etc. A default database (named `'$'`) is used for operations without an explicit database argument. The default database is shared i.e. when no definition exists in a named database, calls are redirected to predicate definitions in the default database.

The state of an agent (seen as a set of dynamic Prolog clauses) is contained entirely in a database with a name derived from the name of the agent.

LeanProlog also supports dynamically created databases controlled through unique handles allowing agents to hide private state information.

## 2.2 A "Twitter-style" agent inheritance mechanism

Agents share compiled code – which is immutable in LeanProlog. They also inherit code automatically from the default dynamic database of a given agent space when calling predicates for which they do not provide their own definitions.

An agent can decide to "follow" a set of other agents. The set of agents followed by a given agent can be updated dynamically at any time. Syntactically, running the goal

```
?-cindy@[alice,bob].
```

ensures that agent `cindy` follows agents `alice` and `bob` but later `cindy` can change her mind and issue

```
?-cindy@[alice,dylan].
```

from which point in time `cindy` follows a different set of agents.

This results in a fully dynamic, but a weak, non-transitive, strictly "one-level" inheritance mechanism between databases.

Note also that an agent inherits a *complete predicate definition* from the first of the followed agents that provides it. This choice is justified, as inheriting different fragments of a predicate from various agents would result in programs difficult to debug. Moreover, an agent can only inherit code present in the space where it runs a given goal, i.e. it inherits either from local agents or from agents visiting the agent space where its code executes.

This mechanism is a radical departure from typical object oriented languages. Our decision is motivated by the fact that the dynamic nature of an agent's agreement to follow other agents would quickly create unexpected consequences if propagated to its followers.

This inheritance mechanism is implemented by adapting the traditional Prolog meta-interpreter handling dynamic code to look-up dynamic predicate definitions for a given agent provided by the agents it follows, as well as shared code in the default database `'$'`.

### 2.3 Agent Interactions

Communication between agents located in the same agent space, is achieved through dynamic database updates or predicate calls as well as through code inherited from the followed agents. We illustrate how these mechanisms work with the predicate `local_agent_test`.

```
local_agent_test:-
  assert(friends(cool_people)),
  alice@[bob,cindy], % alice follows bob and cindy
  alice@assert(like(macs)),
  alice@assert(like(popcorn)),
  alice@assert(hate(candy)),
  alice@((hate(pcs):-true)), % shorthand for assert
  cindy@[alice,bob], % cindy starts following alice
  bob@((like(X):-alice@hate(X))),  % bob likes what alice hates
  foreach(cindy@friends(X),println(friends:X)),
  foreach(bob@like(X),println(bob:likes(X))),
  foreach(alice@like(X),println(alice:likes(X))),
  foreach(cindy@hate(X),println(cindy:hates(X))).
```

When running the predicate one can observe the fairly natural semantics of "following" another agent, for instance that `bob` likes candy because `cindy` hates it.

```
?- local_agent_test.
friends : cool_people
bob : likes(candy)
bob : likes(pcs)
alice : likes(macs)
alice : likes(popcorn)
cindy : hates(candy)
cindy : hates(pcs)
true.
```

Cooperative coordination patterns, as described in [12], or high-level multi-threading encapsulated in parallel `fold` and `findall` operations as described in [11] are available to express various interaction protocols.

The following example illustrates how one can emulate message exchanges between agents coordinated cooperatively using the Linda protocol described in [12].

```
coop_mes_test(N):-
  C='$mes_coord',
  alice@[],bob@[],cindy@[],
  new_coordinator(C),
  N2 is N*2,
  new_task(C, (
     for(_,1,N2),
       alice@handle_msg(From,m(I)),
       println(alice_got(From,m(I)))
    )
  ),
  new_task(C, (
    for(I,1,N),
      println(bob_sent(alice,m(I))),
      bob@send_msg(alice,m(I))
    )
  ),
  new_task(C,(
    for(I,1,N),
      println(cindy_sent(alice,m(I))),
      cindy@send_msg(alice,m(I))
    )
  ),
  coordinate(C),
  stop_coordinator(C).
```

Note that multi-threading is avoided in this case and the actual execution is handled by the predicate `coordinate/1` that manages the protocol cooperatively implemented in terms of operations on first-class logic engines. Such protocols have the advantage to work deterministically and significantly faster than their multi-threaded equivalents due to reduced scheduling, memory allocation and synchronization costs.

The following execution trace illustrates the actual message exchanges, happening in a natural order.

```
?- coop_mes_test.
...
bob_sent(alice, m(1))
cindy_sent(alice, m(1))
alice_got(bob, m(1))
bob_sent(alice, m(2))
cindy_sent(alice, m(2))
alice_got(cindy, m(1))
bob_sent(alice, m(3))
cindy_sent(alice, m(3))
alice_got(bob, m(2))
alice_got(cindy, m(2))
alice_got(bob, m(3))
alice_got(cindy, m(3))
```

We will show later that it makes sense to uncouple the execution of logic engines running a given agent's code from the state of the agent. Toward this end we will show next how to uncouple an agent's communication mechanisms and the physical location where its code executes.

## 3 Calling Remote Predicates

The first steps towards distributed execution and agent mobility involve designing a reliable and as high level as possible remote execution mechanism.

LeanProlog uses Java's *Remote Method Invocation* (RMI) communication layer to connect Prolog processes on the same or different machines together. Our Prolog-level communication layer, called *Remote Logic Invocation* (RLI) consists of a mechanism to automatically start a registry containing the names (called *ports* in RMI parlance) of the Prolog processes on a given host, as well as a server and a client API designed under the assumption that calls are *atomic* and guaranteed to terminate.

On the Java side, the following interface specifies the operations that a client can request from a server:

```
package rli;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface ServerStub extends Remote {
  public Object rli_in() throws RemoteException;

  public void rli_out(Object T) throws RemoteException;

  public int rli_ping() throws RemoteException;

  public int rli_stop_server() throws RemoteException;

  public Object rli_call(Object T) throws RemoteException;
}
```

Among them, `rli_in` and `rli_out` can be used to implement message queues, `rli_ping` can test if a server is alive and `rli_stop_server` can be used to shut down a server remotely. However, the only operation relevant for our agent layer is `rli_call` which specifies a deterministic (first solution) call to a remote Prolog goal. On the Prolog side a client executes

```
rli_call(Host,Port,Goal,Result).
```

To start a server on a named Port (and possibly activate the RMI registry keeping track of servers locations and capabilities, unless it is already running) one executes:

```
rli_start_server(Port).
```

A few additional predicates are available to handle message queues or to let a client to wait until a server is running, but as we will show in the next sections these two predicates are all we need for implementing high-level mechanisms that support agent mobility and remote agent communication. Of special interest are rli_start_broker and rli_register(Host, ThisHost, ThisPort) that start a special well-known server – called a "broker" – which maintains a registry of all agents on a given host, together with their location information.

## 4 Agent Spaces

An *agent space* is seen as a container for a group of agents usually associated with a Prolog process and an RLI server. To keep things simple we can assume that the name of the space is nothing but the name of the RLI port.

The predicate start_space(BrokerHost,ThisHost,Port) starts, if needed, the unique RLI service associated to a space and registers it. Registration happens after making sure that on each host, a broker, keeping track of various agents and their homes, is started, when needed.

```
start_space(_,_,Port):-
  '$space' ⟹ OtherPort,
  !,
  errmes(no_space_started_at(Port),already_started_space(OtherPort)).
start_space(BrokerHost,ThisHost,Port):-
  '$space' ⟸ Port,
  ( BrokerHost == localhost → rli_start_broker
  ; true
  ),
  rli_wait(BrokerHost,broker),
  rli_start_server(Port),
  rli_wait(Port),
  rli_register(BrokerHost,ThisHost,Port).
```

Note that a unique global variable (assigned with '==>' in LeanProlog) keeps track of the port associated to an agent space.

Communication with agents inhabiting and agent space happens through this unique port - typically one per process. This requires that all RLI calls to a given port are atomic and terminating.

## 5 Agent Mobility and State Synchronization

Our concept of agent mobility is derived directly from the unique nature of a Prolog program - that can be seen as a set of predicate definitions built each from an ordered set of clauses. This ensures that changes of an agent's state can be safely propagated from an agent space to another without the need to spawn a thread for each otherwise possibly non-terminating remote procedure call. We will now show that it is possible to keep such calls always local using code replication.

## 5.1 Visiting an Agent Space

An agent can *visit* one or more agent spaces at a given time. When calling the predicate `visit(Agent,Host,Port)` an agent broadcasts its database and promises to broadcast its future updates. Note that information about visiting agents is recorded both at their home spaces and the agent spaces they visit. One can conceptualize that an agent is represented at a remote space by a replica of its set of clauses that we call its "avatar". To keep things simple, the database associated to an avatar bears the same name as the the one in its home space.

```
visit(Agent,Host,Port):-
  visiting(Agent,Host,Port),
  !,
  errmes(no_reason_to_visit,already_visiting(Host,Port)).
visit(_Agent,Host,Port):-
  get_space_name(Port),
  Host==localhost,
  !,
  errmes(no_reason_to_visit,already_at(Port)).
visit(Agent,Host,Port):-
  rli_ping(Host,Port),
  assert('$visiting'(Agent,Host,Port)),
  rli_call(Host,Port,(done:-Agent<=='$visiting'),_),
  take_my_clauses(Agent,Host,Port).
```

The predicate `take_my_clauses(Agent,Host,Port)` remotely asserts the agent's clauses to the database of the agent's "avatar". Note that only the agent's own code goes and not the code that the agent inherits locally. This allows context dependent behavior when the code of the avatar is called remotely - for instance, while the reasoning rules about geographical location may travel unchanged, the actual GPS coordinates to which they refer (say in the form of a Prolog fact), may be different at the location the agent visits from those at the agent's home.

As the agent keeps track of all the locations where it has dispatched avatars, it will be able to propagate updates to its database using atomic, guaranteed to terminate RLI calls.

An agent is also able to `unvisit` a given space - in which case the code of the avatar is completely removed and broadcasts of updates to the unvisited space are disabled.

```
unvisit(Agent,Host,Port):-
  retract('$visiting'(Agent,Host,Port)),
  rli_call(Host,Port,gvar_remove(Agent),_),
  rli_call(Host,Port,db_clear(Agent),_).
```

An agent can have followers in various spaces that it visits. Note however that followers inherit the code of the avatar - and therefore all their calls stay local. This also makes more sense as, for instance, an agent asked to find neighboring gas stations should do it based on the GPS location of the agent space it is visiting.

It is an agent's autonomous decision to visit a given agent space. At the same time it is an agent's autonomous decision to become a follower locally or mediated through an avatar. The presence of "free will" on both sides provides flexibility and enables implementation of "anthropomorphic" mechanisms for negotiation, reputation building and trust between agents in a given application.

Fig. 1 sees *agents* as sets of logic engines running on one or more threads in the context of larger and larger distributed computational units. An *agent space* is typically associated to a Prolog process.

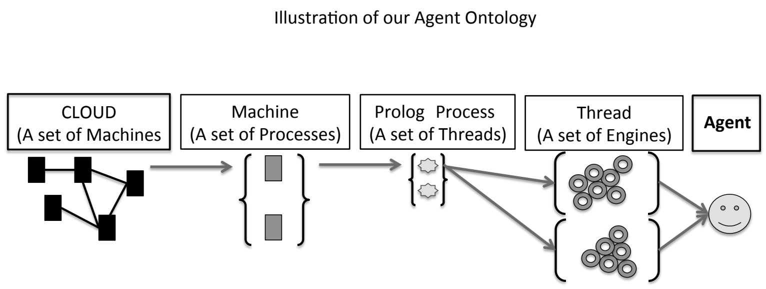
Illustration of our Agent Ontology
CLOUD (A set of Machines)
Machine (A set of Processes)
Prolog Process (A set of Threads)
Thread (A set of Engines)
Agent

Fig. 1: A hierarchical view of agent aggregations

## 6 Distributed Agent Interactions

We will now work through a simple example involving two agent spaces and visiting agents.

First we define a predicate `french_space` that initializes an agent space where salutations occur in French, inhabited by agent `alice`.

```
french_space:-
  start_space(french_space),
  assert(when_arriving_say(bonjour)),
  assert(when_leaving_say(aurevoir)),
  rli_wait(english_space),
  sleep(3), % assuming bob has arrived by now
  alice@[bob],
  alice@salutations,
  alice@visit(english_space).
```

Next we define a predicate `english_space` where salutations occur in English, inhabited by agent `bob`.

```
english_space:-
  start_space(english_space),
  assert(when_arriving_say(hello)),
  assert(when_leaving_say(goodbye)),
  bob@[],
  bob@visit(french_space),
  bob@((salutations:-
        when_arriving_say(A),
        println(A),
        sleep(5), % we can do something more interesting here
        when_leaving_say(B),
        println(B))),
  sleep(3),
  bob@unvisit(french_space),
  % assuming bob expects that alice is visiting
  alice@[bob],
  alice@salutations.
```

When running the two predicates in two different terminal windows one can see the results of `bob` visiting `french_space`.

```
?- french_space.
bonjour
aurevoir
```

and the results of `alice` visiting `english_space`.

```
?- english_space.
hello
goodbye
```

The example illustrates the semantics of "visiting". When `bob` visits, he promises that future code updates will follow him - resulting in the predicate `salutations` being brought to `french_space`. Since `alice` follows him this predicate becomes available to her there. Note that the salutation messages come out in French, given that bob's predicate depends on the local facts `when_arriving_say/1` and `when_leaving_say/1`.

When `alice` visits `english_space` and decides to follow `bob` there, her salutations come out in English. Note also that `alice` gains access to the predicate `salutations` simply by following `bob`, but that she needs to state that in every space where she agrees to do so. This illustrates the workings of an agent's "free-will" and one can use such mechanisms for implementing complex negotiation protocols.

## 7 Related work

There's a significant number of papers relating agents and distributed programming, going back as early as [13] and [14] and covering agent interaction as well as various aspects of knowledge sharing. Combining concurrency with a shared database has been described in a multi-threaded implementation in various Prolog systems as early as in in [15].

There are several agent toolkits based on Prolog. Examples include Tu-Prolog [16] which can use .NET or Java RMI, LPA Chimera Agents toolkit running on the Windows platform, Sicstus Prolog that provides agent support using add-ons for TCP/IP, .NET and Java interfaces for distribution, the SRI Open Agent Architecture based on either of Sicstus or Quintus Prolog, and Qu-Prolog that uses a middleware broker system called "Pedro" handling TCP/IP based inter-process messaging. The SRI Open Agent Architecture (OAA) [17] is also written in Prolog and uses a remote predicate call based messaging protocol, but has a different philosophy of layering, and methods for the formation of agent societies. SOCS agents are based on Sicstus Prolog with JXTA and XML messaging, which is a heavyweight solution in our opinion, but the authors acknowledge that at the time there were few choices for technologies available to build dynamical multi-agent societies.

The closest implementation to our system is Tu-Prolog [16] which also provides a high level agent layer using objet oriented abstractions. It also sees workspaces as logical containers, used to structure the overall environment where the agents play. As in most mobile agent implementations Tu-Prolog agents can dynamically join and concurrently work in multiple workspaces. Among the differences our agent mobility protocol and the "Twitter-style" code inheritance mechanism.

Many agent toolkits are available in procedural programming languages that lack the ease of representation and reasoning models that characterize declarative approaches, and none of them support native high level agent architectures that require little knowledge of the programmer to create networks of societies of agents. Examples of such procedural tools are ABLE [18], JADE [19] and JACK [20], which can be connected to Prolog systems using middleware approaches.

In this context, our emphasis has been on high-level design constructs that not only relieve the programmer of a lot of burdensome implementation efforts, but that enable rapid prototyping of various computational models. For example, inter-agent communication based on message-passing can also be built by creating Agents as specialized message-brokers, between LeanProlog spaces. Agents as message-brokers combine the dynamic addressing and routing of mobile-mailboxes: the transmission of agents as message-brokers by visiting spaces and their subsequent use in processes where they were not previous found enables one to create massively parallel, dynamically altering patterns of interconnections between communicating LeanProlog machines. Our agent model supports non-deterministic interactions between multiple agents that may be distributed internet-wide, for instance in multi-agent logistics and planning problems.

Other issues that some agent platforms run into in real world applications are that with large numbers of agents, one rapidly runs out of the TCP/IP address spaces when one TCP/IP port is used per agent. Also, the overheads in programming and maintenance are much higher as a system evolves because one has to artificially build namespaces and mechanisms to keep track of them, which, in a system that is designed to evolve dynamically, is very difficult to do.

## 8 Conclusion

We have described a distributed multi-agent architecture that, despite its simplicity exhibits some novelty in terms of the way agents inherit dynamic code and the way they engage in communication with other agents. Our concept of agent mobility is based on a simple remote logic invocation mechanism. While quite straightforward – by limiting calls to database updates – it provides remote code sharing without requiring potentially non-terminating remote predicate calls and it allows a "free will" mechanism that agents can exercise when using other agents' knowledge bases. We have deliberately not covered various reasoning mechanisms that agents can implement - these are seen as independent of the infrastructure itself - our main focus in this paper.

## Acknowledgment

Paul Tarau's work has been supported in part by NSF (research grant 1018172) and Vivomind Research LLC. Arun Majumdar thanks VivoMind Research LLC for support.

## References

1. Tarau, P.: Towards Inference and Computation Mobility: The Jinni Experiment. In Dix, J., Furbach, U., eds.: Proceedings of JELIA'98, LNAI 1489, Dagstuhl, Germany, Springer (October 1998) 385–390 invited talk.
2. Tarau, P.: Intelligent Mobile Agent Programming at the Intersection of Java and Prolog. In: Proceedings of The Fourth International Conference on The Practical Application of Intelligent Agents and Multi-Agents, London, U.K. (1999) 109–123
3. Tarau, P.: Inference and Computation Mobility with Jinni. In Apt, K., Marek, V., Truszczynski, M., eds.: The Logic Programming Paradigm: a 25 Year Perspective, Springer (1999) 33–48 ISBN 3-540-65463-1.
4. Tarau, P.: Agent Oriented Logic Programming Constructs in Jinni 2004. In Demoen, B., Lifschitz, V., eds.: Logic Programming, 20-th International Conference, ICLP 2004, Saint-Malo, France, Springer, LNCS 3132 (September 2004) 477–478

5. Mayfield, J., Labrou, Y., Finin, T.W.: Evaluation of KQML as an Agent Communication Language. In Wooldridge, M., Müller, J.P., Tambe, M., eds.: ATAL. Volume 1037 of Lecture Notes in Computer Science., Springer (1995) 347–360
6. FIPA: FIPA 97 specification part 2: Agent communication language (October 1997) Version 2.0.
7. Hermenegildo, M.V.: An abstract machine for restricted AND-parallel execution of logic programs. In: Proceedings on Third international conference on logic programming, New York, NY, USA, Springer-Verlag New York, Inc. (1986) 25–39
8. Lusk, E., Mudambi, S., Gmbh, E., Overbeek, R.: Applications of the Aurora Parallel Prolog System to Computational Molecular Biology. In: In Proc. of the JICSLP'92 Post-Conference Joint Workshop on Distributed and Parallel Implementations of Logic Programming Systems, Washington DC, MIT Press (1993)
9. Tarau, P.: Fluents: A Refactoring of Prolog for Uniform Reflection and Interoperation with External Objects. In Lloyd, J., ed.: Computational Logic–CL 2000: First International Conference, London, UK (July 2000) LNCS 1861, Springer-Verlag.
10. Tarau, P., Majumdar, A.: Interoperating Logic Engines. In: Practical Aspects of Declarative Languages, 11th International Symposium, PADL 2009, Savannah, Georgia, Springer, LNCS 5418 (January 2009) 137–151
11. Tarau, P.: Concurrent Programming Constructs in Multi-engine Prolog: Parallelism just for the cores (and not more!). In Carro, M., Reppy, J.H., eds.: DAMP, ACM (2011) 55–64
12. Tarau, P.: Coordination and Concurrency in Multi-engine Prolog. In Meuter, W.D., Roman, G.C., eds.: COORDINATION. Volume 6721 of Lecture Notes in Computer Science., Springer (June 2011) 157–171
13. Agha, G.: ACTORS: A Model of Concurrent Computation in Distributed Systems. The MIT Press: Cambridge, MA (1986)
14. Halpern, J.Y., Moses, Y.O.: Knowledge and common knowledge in distributed environments. In: Proceedings of the 3rd ACM Conference on Principles of Distributed Computing, ACM Press (1984)
15. Carro, M., Hermenegildo, M.V.: Concurrency in Prolog Using Threads and a Shared Database. In: ICLP. (1999) 320–334
16. Ricci, A., Viroli, M., Piancastelli, G.: simpa: An agent-oriented approach for programming concurrent applications on top of java. Sci. Comput. Program. **76** (January 2011) 37–62
17. Cheyer, A., Martin, D.: The open agent architecture. Autonomous Agents and Multi-Agent Systems **4** (March 2001) 143–148
18. Bigus, J.P., Schlosnagle, D.A., Pilgrim, J.R., Mills, W.N., Diao, Y.: Able: a toolkit for building multiagent autonomic systems. IBM Syst. J. **41** (July 2002) 350–371
19. Spanoudakis, N.I., Moraitis, P.: Modular JADE Agents Design and Implementation Using ASEME. In Huang, J.X., Ghorbani, A.A., Hacid, M.S., Yamaguchi, T., eds.: IAT, IEEE Computer Society Press (2010) 221–228
20. Tweedale, J., Ichalkaranje, N., Sioutis, C., Jarvis, B., Consoli, A., Phillips-Wren, G.: Innovations in multi-agent systems. Journal of Network and Computer Applications **30**(3) (August 2007) 1089–1115