Object Oriented Logic Programming as an Agent Building Infrastructure

Paul Tarau

***Abstract*— We show that agent programming patterns are well expressed in terms of an object oriented layer extended with a generalized inheritance mechanism and independent logic programming based inference engines. Instead of proposing yet another agent programming *"model"* we simply derive key agent programming patterns as the natural result of a set of programming language constructs.**

The suggested equation: *Agents = Objects + Logic + Inference Engines + Coordination + Remote Action* provides orthogonal agent composition mechanisms which are expressive and highly reusable.

The approach described in this paper has emerged in the process of building Agent classes as extensions to our industrial strength Object Oriented Prolog system, Jinni 2002, available for online evaluation at:

`http://www.binnetcorp.com/Jinni`

Keywords: Agent Programming Languages, Remote Execution, Blackboard-based Agent Coordination, Multi-threaded Logic Programming Engines, Distributed AI

I. Introduction

The paradigm shift towards networked, mobile, ubiquitous computing has brought a number of challenges which require new ways to deal with increasingly complex patterns of interaction: autonomous, reactive and mobile computational entities are needed to take care of unforeseen problems, to optimize the flow of communication, to offer a simplified and personalized view to end users. These requirements naturally lead towards the emergence of *agent programs* with increasingly sophisticated inference capabilities, as well as autonomy and self-reliance.

Jinni (Java INference engine and Networked Interactor) [1], [2], is a lightweight, multi-threaded compiled Prolog system with powerful Object and Agent Oriented extensions, intended to be used as a flexible scripting tool for gluing together knowledge processors with Java and .NET components in distributed applications. Jinni's high level, portable, secure networking layer provides intelligent interoperation between Java and .NET components on Windows on Linux/Unix and Pocket PC platforms.

Jinni threads are coordinated through blackboards, local to each process. Associative search based on term unification (a variant of Linda [3], [4]) is used as the basic synchronization mechanism. Threads, blackboard and networking operations are controlled with lightweight, fast bytecode Prolog interpreters. The synergy of these features makes Jinni a convenient development platform for distributed AI, and in particular, for building intelligent autonomous agent applications.

Contact address: **Paul Tarau, Ph.D.** Department of Computer Science, University of North Texas, P.O. Box 311366, Denton, Texas 76203, E-mail: tarau@cs.unt.edu)

II. The Jinni Ontology: Orthogonal Language Constructs for Agent Programming

As technology matures and design patterns emerge and consolidate, Agent Programming is getting closer and closer to a *programming paradigm* status [5]. This implies a high degree of compositionality - ability to put together general purpose programs from simple, reusable building blocks. With this in mind, we will overview here the set of orthogonal components which make up Jinni's *Ontology*.

Objects: provide proven program composition and code reuse mechanisms and allow extension of a libraries of behaviors and knowledge processing components.

Logic: Logic programming provides well understood, resolution based inference mechanisms. Beyond clause selection in the resolution process and generalized parameter passing, unification provides flexible search in message queues and databases.

Inference Engines: execution of multiple independent goals is needed for implementing complex reactive patterns in agent programs. Engines are lightweight and highly autonomous instances of language interpreters - running through various scheduling models - in particular through blackboard coordinated multi-threading.

Coordination: agent coordination can (and should) be separated from the details of agent communication and the agent's computational mechanisms (engines). We suggest coordination through blackboards - databases with intelligent, constraint based search - instead of conventional message passing.

Remote Action: a simple client-server style remote call mechanism is suggested, as a building bloc for various forms of remote action - in particular for supporting remote event propagation. Building blocks for implementing agent security layers are provided as a combination of server side sandboxing, strong cryptography and password controlled server access. Once security is in place, the suggested infrastructure can safely emulate more flexible P2P interaction patterns.

III. Expressing Key Agent Programming Patterns

We will start by outlining how Jinni's programming language constructs cover key agent programming idioms.

Reactive Behavior: We have avoided any form of interleaving "thinking" and "action" stages in the agent programming model itself. The availability of multi-threading, local and remote blackboard based coordination mechanisms, and multiple reentrant language interpreters allows a logical separation

of concerns: inference mechanisms and reaction to events (expressed as patterns waiting for matching blackboard data) are expressed by orthogonal language constructs and can be programmed as loosely coupled components.

Multi-Agent Mechanisms: Building efficient multi-agent systems effectively is ensured by a flexible object extension mechanism (Cyclic Multiple Depth First Inheritance). Complex Agents are built by importing from a library of agent roles and event processors. Traditional inheritance has been confined to trees (simple inheritance) or lattices (multiple inheritance). This contrasts with the dominant information sharing model - the Web - which has an arbitrary directed graph structure. At the same time, the multi-agent context, with components developed by independent programmers, suggests a directed graph model as the inheritance mechanism for agent code. Intuitively, this allows programmers to be aware only of a small set of similar ”neighbors” and be able to safely import roles and behaviors without being aware of the complete component library.

Agent Negotiation: Blackboard based programming provides natural building blocks for agent negotiation, search and market-style result optimizers. Our blackboards are enhanced with ”blackboard constraint processing” - small chunks of additional code to be tested once the ”waiting” pattern has been matched against new data produced by independent local or remote threads.

IV. Design Issues in Object Oriented Prolog

Agent Oriented Programming can be seen as a natural extension to Object Oriented programming - provided that the object system provides appropriate aggregation mechanisms to build and share agent program components.

A. From modules to classes

Prolog module designs have already tried to provide class-like abilities. It makes sense to conceptually reuse their syntax and their handling of state through local databases for a classes.

B. Class and Instance state: fields or assertions?

Prolog’s associative search through dynamic clauses can easily be reused for supporting both class and instance level state. Still, Object Oriented layers built on top of procedural languages have traditionally used fields. Prolog assert/retract and possibly support for multiple dynamic databases already provides a richer set of state management operations that traditional destructive assignment in procedural language bases object oriented systems.

Clearly fields can be easily approximated as dynamic unary predicates with exactly one clause each. On the other hand ”backward compatibility” with traditional Object Oriented languages suggests a special syntax for that. It also make sense to consider them as a backdoor through which one can introduce strong typing to Prolog - without touching the semantics of unification and clause resolution. The idea is to use (like in Java) classes as types, together with basic types (integers, strings, floats) - and impose dynamic type checking on fields (at assignment time) and on predicate arguments (at call time). The natural mapping of existing Prolog code into a typed universe would be that by default, in the absence of syntax, arguments and fields are of the most generic type.

C. The Logic of Inheritance

Inheritance can be seen as a special purpose inference mechanism. A two line transitive closure Prolog predicate does it - so why do we need it altogether in Logic Programming languages? At a closer look, one will notice that inheritance is in fact inference applied to locating methods in sets of methods (classes). In the presence of method overloading and subtyping, the search mechanism can be seen as a restricted form of unification. The only problem is that, oddly enough, logic programs (seen as sets of clauses) have not been considered first order objects in widely used logic programming languages (Prolog in particular). Another issue is that the dominant object oriented programming style is class based. Fairly sophisticated Reflection packages are needed in languages like Java to manipulate classes and instances as first order objects - and it does not happen in a simple and uniform way. On the other hand - a class based design provides well known techniques to handle most of the inheritance related overhead at compile time and a good basis for a (strong) type system. In a conventional object oriented language, a class is simply a collection of methods and fields. On the other hand a logic program is a set of clauses - a more uniform domain. It is convenient however to see them as a set of predicates (sets of clauses sharing the same main head functor) as the semantics of Prolog ensures that control mechanisms like CUT and backtracking are actually confined to clauses within a predicate. This makes a predicate the appropriate unit to implement a method - and it also implies an ”all or nothing” clause inheritance mechanism: a predicate defined in a class will override all clauses for the same predicate defined elsewhere. Anyway, as most Prolog programmers are aware, mixing clauses from different files will result in unexpected semantics and difficult to maintain programs.

Traditional inheritance has been confined to trees (simple inheritance) or lattices (multiple inheritance). This contrasts with the dominant information sharing model - the Web - which has an arbitrary directed graph structure (handled quite well despite its size and growth). While limiting the scope of inheritance in procedural languages makes sense given the presence of side effects, an arbitrary directed graph model is worth trying out in the context of declarative languages endowed with a formally simpler and cleaner semantics. Procedural languages have been unable to reuse their class systems as a mechanism for name spaces - requiring for this purpose additional constructs like Java’s packages. Interestingly enough, the package/namespace system, as well as the mapping from classes to their member fields and method arguments, form arbitrary directed graphs. As such, they require additional ”language ontology” (declarations, local names, delegation) that breaks the automation induced by free propagation through inheritance. In fact, such constructs are just reformulations of the conventional procedure call based code reuse.

With this in mind, cyclical multiple inheritance looks like a natural choice for designing an object oriented structuring mechanism around a logic programming language. Depth first

search for a matching predicate can happen at compile time - together with a loop checking mechanism. Classes that are parts of a cycle will see their own methods prevailing over methods (predicates) defined elsewhere. A main, prevailing inheritance path based on what's listed first in a file gives (most of) the benefits of single hierarchical inheritance.

Instance and class level state is implemented naturally through local dynamic database mechanisms - with the same "one predicate at a time" assumption: an instance level dynamic predicate will replace all clauses of a class level predicate if overriding is used.

V. Classes, Instances and Inheritance in Jinni's Object Oriented Prolog

Jinni 2002 features a simple, elegant and fast Object Oriented Prolog layer - built as a natural extension to ISO Prolog. Classes are just Prolog files with include declarations - almost no changes are required to reorganize your existing Prolog code in an Object Oriented style. As the dispatching of method calls is handled at compile time and instances are lightweight, Jinni 2002's Prolog Objects are extremely efficient. Prolog class files can be located at arbitrary URLs on the Web - one can inherit and override from a virtually unlimited library of existing Prolog files. And what if cycles will form? Not a problem! Jinni 2002 supports multiple cyclic inheritance for building in a scalable way, an arbitrary network of Web or file based interdependent Prolog classes!

A class foo is associated to each Prolog file (or URL) let's say foo.pl. In a way similar with compiling a Prolog file in a conventional way with `?-compile(foo)`, the user can enter a default instance of the class foo with

```
?-enter_class(foo).
```

or call code in it (like in the presence of a conventional module system) from outside with

```
?-foo:<predicate>.
```

If foo.pl contains `:-[<superclassfile>]` declarations, let's say something like `:-[bar]`, definitions not found in foo will be searched in bar. As files and URLs are treated in similar ways, inheritance directives like `:-['http://www.my_url.com']` can refer to non-local URLs as well.

ISO Prolog's `:-initialization((<Goal>))` declarations are processed (in reverse order) after collecting them from included files - as a mechanism of class level initialization, shared among all instances. Code added with **assert/1** in such **initialization/1** calls will be visible in all instances.

Code inheritance is handled at compile time - through a special `ocompile/1` command which reinterprets include directives like `:-[<file>]` as inheritance from other Prolog files/classes. Classes are compiled on the fly, at the first use of a class or creation of a new instance.

The multiple cyclical depth first inheritance mechanism is implemented by keeping the path consisting of the list of visited includes, when (at compile time) predicates not defined locally, are brought from files or URLs. In the presence of multiple includes, a "depth-first" order for finding definitions ensures that a dominant main inheritance tree prevails in case of ambiguity. This concept of cyclical inheritance allows reuse of Prolog code located virtually everywhere on the Web from a local perspective.

Constructors are simply predicates of various arities having the same name as the file (or URL). At instance creation time, no-arg constructors of supers are automatically called, in reverse inheritance order. This usually ensures that fields defined locally or in the main inheritance chain will prevail. Programmers should initialize most fields in these default constructors. Constructors (of any arity) are inherited, if not provided. They can freely call constructors defined in their super classes using their predicate names, same as the ones of the superclasses' names.

Instance Fields are local to each instance. A **set** operation `<field name> <= <value>` and a **get** operation `<field name> => <Prolog variable>` are provided, like in:

```
radius <= 99
radius => R
```

Assert/1 and other database operations are local to instances - but the results of class level asserts are visible in instances - until overridden by a local assert with the same predicate name and arity.

Instances are created from **Constructors** (same as class names if no-arg, having the main functor the same as the class name if having arguments) with the **new/2** command:

```
new(Constructor,Instance)
```

In the process, an internal **Class** handle is created and if needed the code for the class is compiled on the fly and attached to it.

Class Fields are shared among instances. In fact, they are just instance fields belonging to instance 0 - which keeps the state of the class itself. A class field set operation `<field name> <== <value>` and a class field get operation `<field name> ==> <Prolog variable>` are used as in:

```
instance_count <== 10
instance_count ==> R
```

When performed from instances of a given class, these operations are applied to the fields of the class, not the instance. Like static variables in Java, they refer to data shared among all instances of a class.

VI. Other Agent Building Blocks: Engines, Threads and Blackboards

A. Multiple Inference Engines, Answer Generation and Control

Independently of its multi-threading mechanism, Jinni 2002 provides first order inference engines - separate instances of its dynamically growing/shrinking runtime system (heap,stack,trail) which can be controlled through the following API:

- **new_engine(Instance, AnswerPattern, Goal, EngineHandle):** creates and returns a new engine based on code

and dynamic database state associated with a Prolog class instance

- **get(EngineHandle, Answer):** asks an engine for a new Answer, which will be of the form **the(AnswerPatternInstance)** on success and which will be no on failure as well on any call after failure occurred
- **stop(EngineHandle):** makes sure the engine is stopped. Only no answers will be available from the engine in the future.
- **return(Answer):** initiated by the engine - which acts such that the next **get/2** of the parent will obtain a copy of Answer. Note that engines are fully reentrant - in particular, the parent can force the engine to resume its work with another **get/2 request** - in which case the engine performs as if the **return/1** statement were not in effect.

Example:

```
?- new_engine(X,(member(X,[1,2]),
   (X=1,return(good(X));X>1)),E),
   get(E,A),get(E,B),get(E,C),get(E,D).

A = the(good(1)) B = the(1) C = the(2)
D = no E = 1331 X = _120 ;
```

This kind of functionality seems the simplest way to implement some agent-style control mechanism on top of Prolog - it provides a minimal set of language constructs for the kind of control users want to have interactively over Prolog's answer production.

B. Threads and Hubs

Jinni 2002 supports a simple multi-threading model, given by the following API:

- **bg(Goal,ThreadHandle):** launches a new thread executing Goal and returns a ThreadHandle to it
- **hub_ms(Timeout,HubHandle):** constructs a new Hub returned as a HubHandle - a synchronization device on which **N** consumer threads can wait with **collect(HubHandle,Data)** for data produced by **M** producers providing data with **put(HubHandle,Data)**. However, if a given consumer waits more than Timeout milliseconds it returns and signals failure. As usual in Java, 0 timeout means indefinite suspension.
- **current_thread(ThreadHandle):** returns a handle to the current thread - might be passed to another thread wanting to join this one.
- bf join_thread(ThreadHandle): waits until a given thread terminates.
- **sleep_ms(Timeout):** suspends for Timeout milliseconds, while consuming minimal (practically no) CPU power

C. Thread Coordination with Blackboards

Blackboards are global (one per Jinni process) databases which provide thread coordination through the following (extended Linda) operations:

- **in(Pattern):** waits for data on the blackboard which matches (through unification) Pattern
- **out(Pattern):** puts for data on the blackboard and possibly resumes a thread waiting with **in/1** if Pattern matches
- **all(Pattern,Matches):** returns a list of terms on the blackboard ready to match Pattern or an empty list if none matches. Such data has been put on the blackboard using **out/1** operations.
- **wait_for(Term,Constraint):** waits for a term on the blackboard, such that Constraint holds
- **notify_about(Term):** notifies a suspended matching **wait_for(Term,Constraint)**, if **Constraint** holds, that **Term** is available

Blackboard operations can be combined with **remote_run** remote predicate calls to allow interaction between threads distributed in different processes on the same or on different computers on the net. Threads can be launched locally or remotely with **bg** operations.

1) Using Blackboard Constraints: The natural extension to Linda [3], [4] introduced in Jinni is to use *constraint* solving for the selection of matching terms, instead of plain unification, as provided by **wait_for(Term,Constraint)** and **notify_about(Term)**.

For instance,

```
notify_about(stock_offer(qqq,21))
```

would trigger execution of a thread having issued

```
wait_for(stock_offer(qqq,Price),Price<22).
```

while something like

```
notify_about(stock_offer(qqq,23))
```

would leave the thread having issued the **wait_for** operation suspended.

Note that in a client/server Linda interaction, triggering an atomic transaction when data verifying a simple arithmetic inequality becomes available, would be expensive. It would require repeatedly taking terms out of the blackboard, through expensive network transfers, and put them back unless the client can verify that a constraint holds. On the other hand, a server side execution checks a constraint only after a match occurs between new incoming data and the head of a suspended thread's constraint checking clause, i.e. a basic indexing mechanism is used to avoid useless computations. In this setting, a remote client thread can perform all the operations atomically on its own thread, using local operations on the server, and return the computed results asynchronously. The (simplified) fragment showing the implementation of `wait_for` and `notify_about` is as follows:

```
wait_for(Pattern,Constraint):-
  if(take_pattern(available_for(Pattern),
                  Constraint),
     true,
     ((
       out(waiting_for(Pattern,Constraint)),
       in(holds_for(Pattern,Constraint))
     ))
  ).

notify_about(Pattern):-
  if(take_pattern(
       waiting_for(Pattern,Constraint),
       Constraint),
     out(holds_for(Pattern,Constraint)),
```

```
    out(available_for(Pattern))
  ).

% takes the first matching Pattern
% for which Constraint holds
take_pattern(Pattern,Constraint):-
  all(Pattern,Ps),
  member(Pattern,Ps),
  call(Constraint),
  in(Pattern,_).
```

Note that each time the head of the waiting clause matches incoming data, its body is (re)-executed.

Although termination of constraint checking is left in the programmer's hand, only one thread is affected by a loop in the code. This is quite important in a multi-agent setting as it ensures that a server's integrity as such not being compromised by a looping client thread. Note that incorporating symbolic constraint reducers (CLP, FD or interval based) can significantly improve performance for large scale problems.

VII. Remote Execution with Object Oriented Prolog Classes

As part of the growing library of Object Oriented Prolog components (see directory classlib), Jinni provides a server and a client class which will be used as inheritance roots for various components (secure client and server, agents etc.).

A. The Server and Client classes

The constructor **server(Port,Password)** creates a server instance listening on a port and only executing queries of clients providing a matching password.

The constructor **client(Host,Port)** creates a client which will try to connect to a server on Host, Port. The resulting client is ready to send queries with **ask(Goal)**. While a client performing a **remote_run** operation opens and closes the socket automatically, instances of this client will keep the socket open until the programmer uses a disconnect operation. All calls on a given socket run on the same thread on the server. This added flexibility requires less system resources and avoids **TCP_WAIT** related errors on operating systems slow in freeing the ports of closed sockets. Note that a password (*tweety* in the examples) is needed for the server to accept a client's requests.

```
Server Window:
?- new(server(8888,tweety),S),
   S:serve.
hello

Client Window
?- new(client(localhost,8888,tweety),
       C),
   C:ask(println(hello)),
   C:disconnect.
```

B. Secure Communications Using Cryptography and Sealed Objects

When running under JDK 1.4 or later, which integrates a cryptography package Jinni provides secure communication classes. Note that from a user's perspective security is completely transparent, as shown in the following example:

```
Server Window:
?- new(secure_server(8888,tweety),S),
   S:serve.
hello

Client Window
?- new(secure_client(localhost,8888,tweety),
       C),
   C:ask(println(hello))
```

Internally, Jinni makes use of serialized Prolog terms which are encrypted as Sealed Objects before being sent over a socket.

C. The Transport Layer

The transport layer is provided as a simple client-server Remote Predicate Call mechanism given by the following API:

- **run_server(Port,Password):** runs a server on a given Port and with given Password (to be matched by connecting client queries)
- **remote_run(Host,Port,Answer,Goal,Passwd,Result):** asks a server waiting on **Host, Port** to execute **Goal** and return a **Result** of the form **the(AnswerInstance)** if the query succeeds or no if it fails. The returned answer instance contains a copy of **Answer** with bindings resulting of the execution of **Goal**. Note that the execution is deterministic and only the first solution is returned.

Note that the transport layer is a replaceable component and can be provided by RMI, SOAP, CORBA, multicast sockets or any other communication mechanism.

VIII. Agent Programming

A. Agent Programming: form Message Passing to Blackboards and Inference Engines

The first thing that strikes someone looking into performatives + message based agent scripts is the tediousness of communication between agents - reminding the colorful chaos and redundancy of a *bazaar* rather that the crisp architectural rules of a *cathedral*[1], where the actors try to explain, argue and negotiate instead of actually getting business done. Often, message passing based agent communication is dominated by frivolous exchanges focusing on the protocol instead of the work to be performed.

Excessive communication is often an indication of limited intelligence and automation. We believe that inferential mechanisms will enable agents to avoid asking each other the obvious - very much like wise people do. This also means that autonomous search mechanisms, associative processing of events and data records is needed.

Our agent infrastructure design is based on the belief that message passing agent programming constructs should evolve into a more structured blackboard based and inference enabled component technology.

[1] The pun resulting from this comparison is related of course to [6] which advocates the contrary for human actors involved in software engineering tasks.

B. Basic agent programming with Jinni

Agents' behaviors are implemented easily in terms of synchronized in/out Linda operations and remote execution. As an example of such functionality, we will describe the use of two simple chat agents, which are part of Jinni's standard library:

a) Window 1: : a reactive channel listener

```
?-listen(fun(_)).
```

b) Window 2: : a selective channel publisher

```
?-talk(fun(jokes)).
```

They implement a front end to Jinni's associative publish/subscribe abilities. The more general pattern `fun(_)` will reach all the users interested in instances of `fun/1`, in particular `fun(jokes)`. However, someone publishing on an unrelated channel e.g. with `?-talk(stocks(nasdaq))`. will not reach fun/1 listeners because stocks(nasdaq) and fun(jokes) channel patterns are not unifiable.

C. Coordinating the Sell/Buy Function for Stock Market Agents

A more realistic stock market agent's buy/sell components look as follows:

```
sell(Who,Stock,AskPrice):-
  % triggers a matching buy transaction on the
  % (remote) server implementing the "market"
  remote_run(
    notify_about(offer(Who,Stock,AskPrice))
  ).

buy(Who,Stock,SellingPrice):-
  % runs as a background thread
  % in parallel with other buy operations
  % while executing code on a remote server
  bg(remote_run(
    try_to_buy(Who,Stock,SellingPrice)
  )).

try_to_buy(Me,Stock,LimitPrice):-
  % this thread connects to a server side
  % constraint and waits until the constraint
  % is solved to true on the server
  % by a corresponding sell transaction
  wait_for(offer(You,Stock,YourPrice),[
    YourPrice=<LimitPrice,
    % server side 'local' in/1
    in(has(You,Stock)),
    in(capital(You,YourCapital)),
    in(capital(Me,MyCapital)),
    MyNewCapital is MyCapital-YourPrice,
    YourNewCapital is YourCapital+YourPrice,
    out(capital(You,YourNewCapital)),
    out(capital(Me,MyNewCapital)),
    out(has(Me,Stock))
  ]).
```

D. Agent Programming with Agent Classes and Inference Engines

Agent classes are built on top of Jinni's Object Oriented Prolog system. Jinni's Multiple Cyclic Inheritance allows static reuse of agent features and agent behavior elements and compositional mechanisms for building new agents from libraries distribute over the Internet.

An Agent Class provides a *goal set* and a *specialized inference engine* working as query interpreter on a separate thread. In a client-server setting this can be seen as a generalized service processor. An agent instance feeds the query interpreter while listening as a server on a port. It also creates a thread for each goal in the goal set. Agent instances have unique global IDs and communicate through remote or local blackboards. Each agent instance runs its own set of goal threads.

```
/*
  Basic Agent Class. Encapsulates
  peer-to-peer capabilities
  (client + server) and
  goal-oriented behavior.
*/

:-[prolog_object].
:-[client].
:-[server].

/*
  Self centered test agent: its client
  component talks to its server component
  on a local port.
*/
agent:-
  agent(behave(20),2000,
        localhost,2000,agatha).

/*
  Default Agent Constructor:
  - runs a goal in background for 20 seconds
  - listens on a local port as a server
  - sets up client side communication with
    a server at a given host and port
*/
agent(Goal,LocalPort,RemoteHost,
                     RemotePort,Password):-
  server(LocalPort,Password),
  client(RemoteHost,RemotePort,Password),
  bg(serve), % starts server thread
  bg(Goal).  % starts goal thread

/*
  Default simple behavior:
  prints messages each second.
*/
behave(N):-
  for(I,1,N),
    sleep(1),
    ask(println(message(I))),
  I=N.
```

Note that the agent class simply a combination of client and server classes together with one or more (background) goal threads.

Deriving an agent using a secure transport layer with the same default behavior is obtained by extending **agent** with **secure_server** and **secure_client** components:

```
:-[secure_server].
:-[secure_client].
:-[agent].
```

where a secure server is defined as:

```
:-[server].
% provides encrypted and
% serialize Prolog terms
```

```
:-[sealed_term].
```

and a secure client is defined as:

```
:-[client].
:-[sealed_term].
```

IX. Related Work

An important number of early software agent applications are described in [7] and, in the context of new generation networking software, in [8], [9].

Mobile code/mobile computation technologies are pioneered by General Magic's Telescript (see [10] for their Java based *mobile agent* product) and IBM's Java based Aglets [11]. Other mobile agent and mobile object related work illustrate the rapid growth of the field: [12], [13], [14], [15], [16], [17].

Implementation technologies for mobile code are studied in [18]. Early work on the Linda coordination framework [3], [19], [20] has shown its potential for coordination of multi-agent systems. The logical modeling and planning aspects of computational Multi-Agent systems have been pioneered by [21], [22], [23], [24], [25], [26], [27], [28].

X. Conclusion

We have shown that agent programming patterns are well expressed in terms of an object oriented layer extended with a generalized inheritance mechanism, by composing library components for secure peer-to-peer agent communication, coordination and goal execution. We have used for this purpose a number of independent programming language constructs like inference engines, threads and remote execution mechanisms. Blackboards with constraints and associative search have been suggested as an alternative to message passing agent architectures. An increasing number of past and ongoing projects are using our agent architecture for applications ranging from virtual personalities to online trading agents and internet based teaching tools. We plan to extend our agent class libraries to cover a larger diversity of agent programming patterns in a number of different application domains.

References

[1] Paul Tarau. Inference and Computation Mobility with Jinni. In K.R. Apt, V.W. Marek, and M. Truszczynski, editors, *The Logic Programming Paradigm: a 25 Year Perspective*, pages 33–48. Springer, 1999. ISBN 3-540-65463-1.

[2] BinNet Corporation. Jinni 2002 A High Performance Java and .NET based Prolog for Object and Agent Oriented Internet Programming. Technical report, BinNet Corp., 2002. Available at http://www.binnetcorp.com/download/jinnidemo/JinniUserGuide.html.

[3] N. Carriero and D. Gelernter. Linda in Context. *CACM*, 32(4):444–458, 1989.

[4] K. De Bosschere and P. Tarau. Blackboard-based Extensions in Prolog. *Software — Practice and Experience*, 26(1):49–69, January 1996.

[5] Y. Shoham. Agent-oriented programming. *Artificial Intelligence*, 60(1):51–92, 1993.

[6] Eric S. Raymond. The Cathedral and the Bazaar. Technical report, www.tuxedo.org, 2002. Available at http://www.tuxedo.org/ esr/writings/cathedral-bazaar.

[7] Jeffrey Bradshaw, editor. *Software Agents*. AAAI Press/MIT Press, Menlo Park, Cal., 1996.

[8] David L. Tennenhouse and David J. Wetherall. Towards an active network architecture. *Computer Communication Review*, 26(2), April 1996.

[9] Gianpaolo Cugola, Carlo Ghezzi, Gian Pietro Picco, and Giovanni Vigna. A characterization of mobility and state distribution in mobile code languages. In *2nd ECOOP Workshop on Mobile Object Systems*, pages 10–19, Linz, Austria, July 1996.

[10] GeneralMagicInc. Odissey. Technical report, 1997. available at http://www.genmagic.com/agents.

[11] IBM. Aglets. Technical report, 1999. http://www.trl.ibm.co.jp/aglets.

[12] Luca Cardelli. Abstractions for Mobile Computation. Technical report, Microsoft Research, April 1999.

[13] Bjarne Steensbaard and Eric Jul. Object and native code thread mobility among heterogeneous computers. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 68–78, Copper Moutain, Co., December 1995.

[14] R. S. Gray. Agent tcl: A flexible and secure mobile agent system. In *Proceedings of the Fourth Annual Tcl/Tk Workshop*, pages 9–23, July 1996. http://www.cs.dartmouth.edu/ agent/papers/tcl96.ps.Z.

[15] Robert S. Gray. Agent tcl: A flexible and secure mobile-agent system. Technical Report PCS-TR98-327, Dartmouth College, Computer Science, Hanover, NH, January 1998. Ph.D. Thesis, June 1997.

[16] D. Eric White. A comparison of mobile agent migration mechanisms. Senior Honors Thesis, Dartmouth College, June 1998.

[17] James E. White. Telescript technology: Mobile agents. In Bradshaw [7]. Also available as General Magic White Paper.

[18] Ali-Reza Adl-Tabatabai, Geoff Langdale, Steven Lucco, and Robert Wahbe. Efficient and Language-independent Mobile Programs. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI)*, pages 127–136, Philadelphia, Pa., May 1996.

[19] S. Castellani and P. Ciancarini. Enhancing Coordination and Modularity Mechanisms for a Language with Objects-as-Multisets. In P. Ciancarini and C. Hankin, editors, *Proc. 1st Int. Conf. on Coordination Models and Languages*, volume 1061 of *LNCS*, pages 89–106, Cesena, Italy, April 1996. Springer.

[20] A. Brogi and P. Ciancarini. The Concurrent Language, Shared Prolog. *ACM Transactions on Programming Languages and Systems*, 13(1):99–123, 1991.

[21] P. R. Cohen and C. R. Perrault. Elements of a Plan Based Theory of Speech Acts. *Cognitive Science*, 3:177–212, 1979.

[22] P. R. Cohen, M. L. Greenberg, D. M. Hart, and A. E. Howe. Trial by Fire: Understanding the Design Requirements for Agents in Complex Environments. *AI Magazine*, 10(3):32–48, 1989.

[23] R. Kowalski and J.-S. Kim. A Metalogic Programming Approach to Multi-Agent Knowledge and Belief. In V. Lifschitz, editor, *AI and Mathematical Theory of Computation: Papers in Honour of John McCarthy*. Academic Press, 1991.

[24] M. Wooldridge. *The Logical Modelling of Computational Multi-Agent Systems*. PhD thesis, Department of Computation, UMIST, Manchester, UK, October 1992. (Also available as Technical Report MMU–DOC–94–01, Department of Computing, Manchester Metropolitan University, Chester St., Manchester, UK).

[25] P. R. Cohen and A. Cheyer. An Open Agent Architecture. In O. Etzioni, editor, *Software Agents — Papers from the 1994 Spring Symposium (Technical Report SS–94–03)*, pages 1–8. AAAIP, March 1994.

[26] P. R. Cohen and H. J. Levesque. Communicative Actions for Artificial Agents. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 65–72, San Francisco, CA, June 1995.

[27] Y. Lésperance, H. J. Levesque, F. Lin, D. Marcu, R. Reiter, and R. B. Scherl. Foundations of a Logical Approach to Agent Programming. In M. Wooldridge, J. P. Müller, and M. Tambe, editors, *Intelligent Agents II (LNAI 1037)*, pages 331–346. Springer-Verlag: Heidelberg, Germany, 1996.

[28] B. Chaib-draa and P. Levesque. Hierarchical Models and Communication in Multi-Agent Environments. In *Proceedings of the Sixth European Workshop on Modelling Autonomous Agents and Multi-Agent Worlds (MAAMAW-94)*, pages 119–134, Odense, Denmark, August 1994.