Orthogonal Language Constructs for Agent Oriented Logic Programming

Paul Tarau

Department of Computer Science and Engineering
University of North Texas
P.O. Box 311366
Denton, Texas 76203
`http://www.cs.unt.edu/~tarau`
tarau@cs.unt.edu

Abstract

We show that key agent programming patterns are well expressed in terms of an object oriented logic programming layer extended with a generalized inheritance mechanism and independent inference engines. Instead of proposing yet another agent programming *"model"* we derive key agent programming patterns as the natural result of a set of programming language constructs providing orthogonal agent composition mechanisms. The approach described in this paper has emerged in the process of building Agent classes as extensions to an Object Oriented Prolog system, Jinni 2004, available for online evaluation at: `http://www.binnetcorp.com/Jinni`.

Keywords: Agents and Logic Programming, Agent Communication Protocols, Agent Programming Constructs, Multi-threaded Prolog Systems, Distributed AI

1 Introduction

The paradigm shift towards networked, mobile, pervasive 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.

In this paper we will overview Jinni 2004's language constructs and infrastructure components [18] used for building agent programs and we will show that the synergy of these features provides an elegant platform for agent applications.

The paper is organized as follows. Section 2 discusses some of the techniques we propose to meet the requirements of a agent programs. Section 3 describes the design of the Object Oriented Prolog layer on top of which Jinni's Agent Programming constructs are built. Other architecture and language level building blocks are discussed in section 4. Section 5 shows how key aspects of agent applications are implemented using Jinni's language and infrastructure constructs. The paper concludes with an overview of related work in section 6, conclusions in section 7 and an `Appendix`, describing a simple multi-agent Mars exploration simulation with Jinni agents.

2 Agent Architectures: a Logical View

We will overview briefly a set of language constructs and architecture elements that we have found instrumental in building Agent Programs effectively.

2.1 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 [15]. This implies a high degree of compositionality - the ability to put together general purpose programs from simple, reusable building blocks.

Objects: provide proven program composition and code reuse mechanisms and allow extension of 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.

2.2 Key Building Blocks for Agent Programming in Jinni

Jinni (**J**ava **IN**ference engine a nd **N**etworked **I**nteractor) [16, 18], is a lightweight, multi-threaded compiled Prolog system with Object and Agent Oriented extensions, intended to be used as a flexible scripting tool for assembling together Prolog, Java and .NET components in distributed applications. Jinni threads are coordinated through blackboards, local to each process. Associative search, based on term unification (a variant of Linda [3, 6]) is used as the basic synchronization mechanism. Threads, blackboard and networking operations are controlled with lightweight, bytecode-compiled Prolog engines.

2.3 Expressing Key Agent Programming Patterns in Jinni

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 of "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. Blackboards are enhanced with "blackboard constraint processing" i.e. small chunks of code to be executed when a "waiting" pattern has been matched against new data produced by independent local or remote threads.

3 Object Oriented Prolog: a Prerequisite of Agent Programming

Agent Oriented Programming can be seen as a natural extension to Object Oriented programming - provided that the object system has appropriate aggregation mechanisms to build and share agent program components such as goals, plans, behaviors and agent communication protocols.

3.1 Designing a Backward Compatible Object Oriented Prolog Layer

3.1.1 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, in the design of classes and instances.

3.1.2 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. 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.

3.1.3 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 types. 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.

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.

3.1.4 Cyclical Multiple Inheritance

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 overriding 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.

3.2 Classes, Instances and Inheritance in Jinni 2004

Jinni 2004's Object Oriented Prolog layer is built as a natural extension to ISO Prolog. Classes are just Prolog files with include declarations - no syntactical changes are required to reorganize existing Prolog code in an Object Oriented style. As the dispatching of method calls is handled at compile time and instances are lightweight, Jinni s Prolog Objects are as efficient as ordinary compiled Prolog code. Prolog class files can be located at arbitrary URLs on the Web allowing inheritance from code written and maintained by several programmers at multiple locations. This, together with Jinni's cyclic inheritance provides a scalable way to put together an arbitrary network of Web or file-based distributed programs.

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 call code in a class - like in the presence of a conventional module system - with

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

If `foo.pl` contains `:-[<superclassfile>]` declarations, let's say something like `:-[bar]`, definitions not found in `foo.pl` will be searched in `bar.pl`. As files and URLs are treated in similar ways, inheritance directives like

```
:-['http://www.my_url.com'].
```

can refer to non-local URLs.

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 cyclical inheritance mechanism allows reuse of Prolog code located virtually everywhere on the Web from a *local* perspective.

Overriding and Overloading Overriding is supported one predicate at a time, rather than by combining clauses from different classes. As in any Prolog system, overloading of predicates with different number of arguments is supported and overloaded predicates can belong to different classes.

Constructors are simply predicates of various arities, having the same name as the file (or URL). At instance creation time, no-argument constructors of super classes are automatically called, in reverse inheritance order. **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. If needed, the code for the class is compiled on the fly and attached to it.

Instance Fields are local to each instance. A **set** operation

```
<field name>  <= <value>
```

and a **get** operation

```
<field name>  => <Prolog variable>
```

are provided. **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.

Class Fields are shared among instances. A class field set operation

```
<field name>  <== <value>
```

and a class field get operation

```
<field name>  ==> <Prolog variable>
```

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

Note that no declarations are required for class or instance fields - they can be seen simply as dynamic, runtime operations matching Jinni's design as a scripting language built on top of Java.

4 Other Agent Building Blocks: Engines, Threads and Blackboards

4.1 Multiple Inference Engines, Answer Generation and Control

Independently of its multi-threading mechanism, Jinni 2004 provides "first class" inference engines - separate instances of its dynamically growing/shrinking runtime system (consisting of a heap, stack and 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. Together with the iterator mechanism returning the next answer, the use of **return/1** at any point in the code of the engine, allows co-routining with the parent, without the use of threads.

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 *iterator* functionality [17] seems the simplest way to implement a minimal set of *"performatives"* for controlling independent Prolog engines. It provides the essential language constructs for allowing users (and their agents) to start, explore and stop a Prolog goal's answer computations.

4.2 Threads and Hubs

Jinni 2004 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, after a given consumer waits more than **Timeout** milliseconds, it returns and signals failure. A 0 timeout leads to indefinite waiting.

- **current_thread(ThreadHandle):** returns a handle to the current thread - might be passed to another thread wanting to join this one.
- **join_thread(ThreadHandle):** waits until a given thread terminates.
- **sleep_ms(Timeout):** suspends for **Timeout** milliseconds, while consuming practically no CPU cycles

This *many-to-many* consumer/producer thread synchronization mechanism, centered around *Hubs*, provides a high level building block for blackboard-based agent coordination. As the salient reader will notice, threads and engines are different constructs, as engines are an analogue of iterators which just happen to provide answers to a Prolog queery at each step. When a given engine is run on a separate thread, hubs or blackboards can be used for communication and coordination with other threads (and in particular with their parents).

4.3 Thread Coordination with Blackboards

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

- **wait_for(Term,Constraint):** like `in(Term)`, waits for a term on the blackboard, such that Constraint holds
- **notify_about(Term):** like `out(Term)`, notifies a matching **wait_for(Term, Constraint)**, if **Constraint** holds, that **Term** is available, and that it can resume execution

Blackboard operations are implamented in terms of threads, hubs and Prolog dynamic database operations and 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.

4.3.1 Using Blackboard Constraints

The natural extension to Linda [3, 6] 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.

Although termination of constraint checking is left in the programmer's hand, only one thread is affected by a loop or indefinite suspension 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 or indefinitely suspended client thread.

4.4 Remote Execution with Object Oriented Prolog Classes

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

4.4.1 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. 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.
```

When running under JDK 1.4 or later, which integrates a cryptography package Jinni provides secure communication classes. Internally, Jinni makes use of serialized Prolog terms which are encrypted as Sealed Objects before being sent over a socket.

4.4.2 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, in particular SSL sockets for added security. Moreover, to support full P2P networking (and in particular, server capability behind firewalls or NAT routers), Jinni 2004 provides built-in secure TCP tunneling (a mechanism to transparently reverse client/server roles).

5 Agent Programming in Jinni: Putting it All Together

We will show in this section how to express key agent programming idioms and build typical agent applications in terms of our language and architecture constructs.

5.1 Agent Programming: form Message Passing to Blackboards and Inference Engines

The first thing that strikes someone looking into traces of 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. Most of the time, message passing-based agent communication is dominated by frivolous exchanges focusing on the protocol instead of the work to be performed. This is not surprising, in fact, 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 thinking, context aware human agents do. This also points out the need for autonomous search mechanisms and associative processing of events and data exchanges.

With this in mind, 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.

5.2 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 distributed 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 provided by a **broker_agent** and communicate through remote or local blackboards. Each agent instance runs its own set of goal threads.

5.2.1 A Simple Self-Centered Agent

The following Jinni class implements a minimalist agent consisting of client, server and goal threads, with a self-centered behavior loop in which the goal component `behave/0` requests through the agent's client component to ask the agent's server component to print out a stream of messages.

[1] The pun resulting from this comparison is related, of course, to [14] which advocates the contrary for human actors involved in software engineering tasks.

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

/*
  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

/*
  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 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 is simply a combination of client and server classes together with one or more (background) goal threads.

5.3 Adding Secure Agent Communication

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` class is defined as:

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

and a `secure_client` class is defined as:

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

5.3.1 Adding Sensors and Actuators

Sensor and actuator components are built by wrapping Java-based event handlers and event generators as Jinni objects, using Jinni's Reflection-based Java interface [21]. Synchronization is provided either directly through **Hubs** or through blackboard operations. Remote sensors and actuators are controlled using Jinni's RPC and dedicated threads.

5.3.2 Connecting to External Information Sources and Sinks

Jinni's simple remote predicate call mechanism (also supporting multicast calls [22], TCP-IP tunneling (see `http://logic.csci.unt.edu:9090`) and HTTP-queries (see `http://logic.csci.unt.edu:8080/wordnet_agent/frame.html`) allows interoperation with a variety of external information sources and sinks. We have implemented extensions supporting the Google API for Web-search and a historical stock-market information analyzer agent which converts historical stock market data provided by Yahoo (see `http://logic.csci.unt.edu:7070/index.html`), to Prolog files, available online for dataminig and machine learning applications. Using a Java-based protocol adaptor we have also implemented a Yahoo IM agent which provides an interface to a knowledge bases conversational agent application [7] easily accessible from wireless PDAs an phones.

5.3.3 Agent Monitoring

Jinni's blackboards can be used for implementing non-intrusive and fault tolerant agent monitoring mechanisms. Various agent components will post periodically status records which will be collected asynchronously by monitoring agents. We have used this mechanism to set up a P2P network of "federated" Web-server agents which are mutually monitoring each other for fault-tolerance and load balancing.

5.3.4 Agents with Continuous Planning

As Jinni agents are not based on a monolithic *sense-plan-act-sense* agent loop, it is possible to easily interleave planning and with reactive loops using blackboard constraints for synchronization. This mechanism allows provides the benefits of continuous planning while confining the the implementation of the planner to a separate component.

5.3.5 Facilitator Agents

A key facilitator is a broker agent which mediates service exchanges among Jinni agents. As Jinni agents are essentially peer-to-peer through their dual client server components facilitator agents are only needed for dispatching unique global IDs and as well-known starting vertices of a P2P agent network. A key element in providing P2P functionality to a set of agents is the ability to "reverse" client-server links at will, using Jinni's built-in TCP-IP tunneling API.

Jinni supports a simple TCP-IP tunneling API to export services provided by servers behind a firewall, in particular Jinni's own Prolog remote predicate call server, or its HTTP server usable to deploy Jinni applets or server side Prolog agents. The API consists of server and client side tunneling components:

- `server_tunnel(ServerPort,LinkServerPort,ServerTunnel)`: Creates a server side TCP-IP tunnel which maps ServerPort to LinkServerPort. When a client connects to `LinkServerPort`, it will be able to map all the services of a server it has access to (usually behind a firewall or NAT) as if they were provided on `ServerPort`. This component is meant to be used on a machine visible on the net to which services behind firewalls can connect and become visible. The returned `ServerTunnel` can be used to stop the (background) threads created to support the tunnel.

- `client_tunnel(LinkHost,LinkPort,LocalServer,LocalPort,ClientTunnel)`: Creates a client side TCP-IP tunnel which maps `LinkHost, LinkPort` to `LocalServer, LocalPort` where a server (usually behind a firewall or NAT) listens. This client connects to `LinkHost, LinkPort`, and remaps the services at `LocalServer, LocalPort` as if they were provided on `ServerPort` on the `LinkHost` machine. This component is meant to be used on a machine invisible on the net (behind firewalls or NAT). The returned `ServerTunnel` can be used to stop the (background) threads created to support the tunnel.

- `stop_tunnel(Tunnel)`: Stops the threads supporting a server or client side tunnel.

We refer to [18] for examples of use of this API, and to the demo at `http://logic.csci.unt.edu:9090` which provides to a Jinni user behind a firewall dynamically allocated virtual server ports.

5.3.6 Building Multi-Agent Systems

Multi-agent architectures are supported through a combination of P2P-connected broker agents which provide unique global IDs to registered agents and TCP-IP tunneling allowing agents behind a firewall or NAT to export services (see [18]). Agents can have all their threads co-located in the same process or distributed on a cluster of Jinni processes. The **Appendix** describes a Multi-Agent simulation for Mars-exploration rover and orbiter agents.

5.4 3D Programming with Jinni3D Agents

Jinni3D is a Java3D-based extension to Jinni providing a simplified, agent-based layer encapsulating most of the functionality of the fairly difficult and complex Java3D programming API. The combination of force-based 3D graph layout and Jinni agents provides a Prolog API with a 1/10 to 1/20 code size ratio for building multi-threaded 3D programs in applications ranging from games to Web-data visualisation.

6 Related Work

An important number of early software agent applications are described in [1] and, in the context of new generation networking software, in [20]. Design and implementation of various Mobile, Reactive and/or Inferential Agent Infrastructures are described in [9, 8, 16, 19]. Early work on the Linda coordination framework [3, 4, 2] 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 [10, 12, 13, 5, 24]. A surprising variety of Object Oriented Logic Programming resources, including links to system implementations, are available at [11] - ranging from Prolog-independent preprocessors like LogTalk to integrated objects systems like in SICStus Prolog.

In this context, our design suggests building an agent programming infrastructure directly on top of Object Oriented Logic Programming systems, rather than within an Agent model (like BDI [12], for instance), through a number of orthogonal programming language constructs, providing compositional agent building blocks.

In comparison with other object oriented Prologs which mimic more closely the syntax and semantics of object oriented constructs built on top of procedural languages, we have designed Jinni's OO layer as a syntactic extension of Prolog module or file inclusion notation combined with a strong from of inheritance (multiple and cyclic). Given that Jinni is used often as a scripting language on top of Java and that Java provides a strongly typed object system, if needed, we have not associated types to Jinni classes and we have not required declarations for object or class fields.

In comparison with Oz/Mozart [23], the key difference is that we chose to extend Java with a Prolog system providing minimalistic object and agent programming constructs rather than building an integrated multi-paradigm language. Jinni's Object and Agent Oriented programming layers are natural extensions to Prolog both syntactically and semantically, designed with dynamic typing in mind and the uses of Jinni as a scripting language for Internet programming. We think that both approaches have distinct academic and pragmatic merits.

7 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. Our orthogonal design for inference mechanisms, objects, threads, coordination and agent programming constructs is highly reusable in for projects involving such constructs, ranging from the Semantic Web to various agent programming infrastructures as well as for various logic based

References

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

[2] A. Brogi and P. Ciancarini. The Concurrent Language, Shared Prolog. *ACM Trans. Prog. Lang. Syst.*, 13(1):99–123, 1991.

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

[4] 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.

[5] 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.

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

[7] E. Figa and P. Tarau. The VISTA Project: An Agent Architecture for Virtual Interactive Storytelling. In N. Braun and U. Spierling, editors, *TIDSE'2003*, Darmstadt, Germany, Mar. 2003.

[8] 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.

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

[10] 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.

[11] P. Moura. Object-Oriented Logic Programming Resources. Technical report, 2002. Website: http://www.uc.pt/oolpr/bibliography.html.

[12] A. S. Rao and M. P. Georgeff. Modeling rational agents within a BDI-architecture. In R. Fikes and E. Sandewall, editors, *Proceedings of Knowledge Representation and Reasoning (KR&R-91)*, pages 473–484. Morgan Kaufmann Publishers: San Mateo, CA, Apr. 1991.

[13] A. S. Rao and M. P. Georgeff. An abstract architecture for rational agents. In C. Rich, W. Swartout, and B. Nebel, editors, *Proceedings of Knowledge Representation and Reasoning (KR&R-92)*, pages 439–449, 1992.

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

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

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

[17] P. Tarau. Fluents: A Refactoring of Prolog for Uniform Reflection and Interoperation with External Objects. In J. Lloyd, editor, *Proceedings of CL'2000*, London, July 2000. LNCS, Springer-Verlag.

[18] P. Tarau. The Jinni 2004 Prolog Compiler: a High Performance Java and .NET based Prolog for Object and Agent Oriented Internet Programming. Technical report, BinNet Corp., 2004. URL: http://www.binnetcorp.com/download/jinnidemo/JinniUserGuide.html.

[19] P. Tarau and V. Dahl. High-Level Networking with Mobile Code and First Order AND-Continuations. *Theory and Practice of Logic Programming*, 1(3):359–380, May 2001. Cambridge University Press.

[20] D. L. Tennenhouse and D. J. Wetherall. Towards an active network architecture. *Computer Communication Review*, 26(2), Apr. 1996.

[21] S. Tyagi and P. Tarau. A Most Specific Method Finding Algorithm for Reflection Based Dynamic Prolog-to-Java Interfaces. In I. Ramakrishan and G. Gupta, editors, *Proceedings of PADL'2001*, Las Vegas, Mar. 2001. Springer-Verlag.

[22] S. Tyagi, P. Tarau, and A. Mikler. Multicast Protocols for Jinni Agents. In T. Boehme and H. Unger, editors, *Innovative Internet Computing Systems, IICS 2001*, pages 1–18, Ilmenau, Germany, June 2001. Springer, LNAI 2060.

[23] P. Van Roy and S. Haridi. Mozart: A programming system for agent applications. In *International Workshop on Distributed and Internet Programming with Logic and Constraint Languages*, Nov. 1999. Part of International Conference on Logic Programming (ICLP 99).

[24] M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.

Appendix: Jinni 2004 Prototype of an Agent Based Mars Exploration Simulator

The simulator consists of autonomous agents with emphasis on reaction to events and communication relaying.

```
% mission_control.pl: emulates an Earth-based monitoring station
:-[agent].
:-[connections].

mission_control:-
  where(orbiter,OHost,OPort,Pwd),
  where(mission_control,_,MPort,Pwd),
  agent(start,MPort,OHost,OPort,Pwd).

start:-
  println(starting(mission_control)),sleep(1),
  bg(monitor).

monitor:-
  beats(N,T),
  for(I,1,N),
    sleep(T),report(mission_control_ok(I)),
  I=N.

report(Mes):-println(Mes).

% mars_events.pl: emulates external events
:-[server].
:-[connections].

mars_events:-
  where(mars_events,_Host,Port,Pwd),
  bg((server(Port,Pwd),serve)),
  bg(start).

start:-
  report(starting(mars_events)),
  beats(N,T),
  for(I,1,N),
    sleep(T),report(mars_event(I)),
```

```
  I=N.

report(Mes):-println(Mes).

% orbiter.pl - proxy agent connecting mission_control and rovers
:-[agent].
:-[connections].

orbiter:-
  where(mission_control,MHost,MPort,Pwd),
  where(orbiter,_OHost,OPort,Pwd),
  agent(start,OPort,MHost,MPort,Pwd).

start:-
  println(starting(orbiter)),
  sleep(3),
  bg(report).

report:-
  beats(N,T),
  for(I,1,N),
    sleep(T),report(orbiter_ok(I)),
  I=N.

% proxy: communication forwarding
report(Mes):-ask(report(Mes)).

% rover.pl Mars rover with sensors and motors
:-[agent].
:-[connections].

rover(Id):-
  id<=Id,x<=0,y<=0,
  where(orbiter,OHost,OPort,Pwd),
  where(rover,_RHost,RPort0,Pwd),
  RPort is RPort0+Id,
  agent(start,RPort,OHost,OPort,Pwd).

start:-sleep(5),bg(roam),report.

report:-
  id=>Id,
  beats(N,T),
  for(I,1,N),
   sleep(T),x=>X,y=>Y,
   report(rover(Id,step(I,x(X),y(Y)))),
  I=N.

report(Mes):-ask(report(Mes)).

roam:-
  beats(N,T),
  for(I,1,N),
    sleep(T),random_move(x),random_move(y),
```

```
  I=N.

random_move(Coord):-
  random_delta(R),
  Coord=>X,NewX is X+R,Coord<=NewX.

random_delta(X):-random(RX),X is (RX mod 3)-1.

% connections.pl: configuration file for various agent connections

/* where(Agent,Host,Port) facts */

where(mission_control,localhost,2001,eureka).
where(mars_events,localhost,2002,eureka).
where(orbiter,localhost,2004,eureka).
where(rover,localhost,2004,eureka).

/* beats(Number, IntervalInTimeUnits): defines simulation length and speed */
beats(5,3).

% script.pl: launching various agents
go:-
  new(mission_control,_MC),
  new(mars_events,_ME),
  new(orbiter,_O),
  new(rover(0),_R0),
  new(rover(1),_R1),
  println(mission_started).
```

Example of output:

```
Starting Jinni 2004, version 9.92
Copyright (C) BinNet Corp. 1999-2004
Support: binnetcorp@binnetcorp.com
Home: http://www.binnetcorp.com

?- script:go.
server_ready_on_port(2001)
starting(mission_control)
server_ready_on_port(2002)
starting(mars_events)
server_ready_on_port(2004)
starting(orbiter)
server_ready_on_port(2004)
server_ready_on_port(2005)
mission_started
yes
?- mars_event(1)
mission_control_ok(1)
mars_event(2)
orbiter_ok(1)
mission_control_ok(2)
rover(0,step(1,x(0),y(0)))
rover(1,step(1,x(0),y(0)))
```

```
mars_event(3)
orbiter_ok(2)
mission_control_ok(3)
rover(0,step(2,x(-2),y(-1)))
rover(1,step(2,x(-2),y(0)))
mars_event(4)
orbiter_ok(3)
mission_control_ok(4)
rover(0,step(3,x(-2),y(0)))
rover(1,step(3,x(-2),y(1)))
mars_event(5)
orbiter_ok(4)
mission_control_ok(5)
rover(0,step(4,x(-3),y(-1)))
rover(1,step(4,x(-2),y(1)))
orbiter_ok(5)
rover(0,step(5,x(-2),y(0)))
rover(1,step(5,x(-1),y(1)))
```