Agent Oriented Logic Programming in Jinni 2004

Paul Tarau
Department of Computer Science and Engineering
University of North Texas
P.O. Box 311366, Denton, Texas 76203
tarau@cs.unt.edu

ABSTRACT

Jinni 2004 [1, 2, 3] (available from http://www.binnetcorp.com/Jinni) expresses various agent programming constructs in terms of an Object Oriented Logic Programming layer implemented on top of a Java-based Prolog compiler. The architecture provides a high degree of compositionality through the use of a small set of orthogonal programming language constructs.

Keywords

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

1. INTRODUCTION

Jinni 2004 has been designed to support the following orthogonal language constructs:

- **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 are provided for implementing complex reactive patterns in agent programs in both co-routining and multi-threaded execution mode.

- **Coordination**: agent coordination is separated from the details of agent communication and the agent’s computational mechanisms (engines). Jinni 2004 provides coordination through blackboards - databases with intelligent, constraint-based search - instead of conventional message passing.

- **Remote Action**: Jinni supports a simple client-server style remote call mechanism as a building block for various forms of remote action.

2. JINNI 2004: A MULTI-THREADED, DISTRIBUTED, OBJECT ORIENTED LOGIC PROGRAMMING SYSTEM

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. Jinni 2004’s use of a Logic Programming layer (Prolog) as an agent scripting language provides these extensions in a natural way.

2.1 Object Oriented Prolog with Cyclical Multiple Inheritance

Inheritance can be seen as a special purpose inference mechanism. 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. With this in mind, **cyclical multiple inheritance** looks like a natural choice for designing an object oriented structuring mechanism around a logic programming language. 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.

2.2 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.

2.3 Thread Coordination with Blackboard Constraints and Remote Execution

Jinni 2004 supports multi-threading and thread synchronization using Hubs - a simple many-to-many consumer/producer mechanism that also provides high level building blocks for blackboard-based agent coordination. Jinni threads can be launched locally or remotely. Note that threads and engines are different constructs, as engines are an analogue of iterators which just happen to provide answers to a Prolog query 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).

2.4 Using Blackboard Constraints

Blackboards are global (one per Jinni process) databases which provide thread coordination. A natural extension to Linda [4], introduced in Jinni 2004, 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,

\[
\text{notify_about(stock_offer('QQQ',21))}
\]

would trigger execution of a thread having issued

\[
\text{wait_for(stock_offer('QQQ',Price),Price<22)}.
\]

while something like

\[
\text{notify_about(stock_offer('QQQ',23))}
\]

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.

3. AGENT PROGRAMMING WITH AGENT CLASSES AND INFERENCE ENGINES

Agent classes are built on top of Jinni’s Object Oriented Prolog system. 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 identities provided by a broker agent and communicate through remote or local blackboards. Each agent instance runs its own set of goal threads. To implement a minimalist agent consisting of client, server and goal threads, with a self-centered behavior loop in which the goal component requests through the agent’s client component to ask the agent’s server component to print out a stream of messages. The agent class is simply a combination of client and server classes together with one or more (background) goal threads. 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.

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 to function as virtual servers behind firewalls.

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 like role-playing games and interactive fiction.

An increasing number of past and ongoing projects are using our agent architecture for applications ranging from virtual personalities and 3D graph visualization 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.

4. REFERENCES


