The Layered Agent Pattern Language

Elizabeth A.Kendall, Chirag V. Pathak, P.V. Murali Krishna, C.B. Suresh
Computer Systems Engineering, Royal Melbourne Institute Of Technology
City Campus, GPO Box 2476V, Melbourne, VIC 3001 AUSTRALIA
email : kendall@rmit.edu.au

1. OVERVIEW

This paper presents a collection of patterns within a pattern language for agent based systems. Agents have appeared in a wide range of applications, including personalized user interfaces, enterprise integration, manufacturing, and business process support. They are viewed as the next significant software abstraction, and it is expected they will become as ubiguitous as graphical user interfaces are today. Agents are still fairly new, so only some of the patterns presented here have had widespread use; others have been uncovered as being useful and reoccurring problems and solutions at RMIT in the Java Application Framework for Intelligent and Mobile Agents (JAFIMA) Project. Following Examples (Section 2), and Context (Section 3), the Layered Agent is discussed in Section 4; other patterns are presented according to their location within this architectural pattern in Sections 5 to 9. For brevity, some of the patterns are presented in a "patlet" form.

2. EXAMPLES

The present estimate is that there are now over ten thousand agent based applications worldwide; most of them deal with electronic commerce over the Internet, but agents are also proposed as the basis for many critical future applications, including the Teledesic satellite communications project. The following illustrate the kinds of problems that agents address:

- *Projects in Europe and Japan [26] have resulted in web- based systems of "travel agents" that represent and negotiate for travel services. These agents migrate to different web sites to handle hotel, transport, and tour reservations for customers.*
- *An agent based air- traffic control system has been installed at Sydney airport. Individual agents represent airplanes, runways, and air traffic controllers and negotiate with each other for services, takeoffs, landings, and flight paths.*
- *PACT [25] is an agent based system for concurrent multidisciplinary engineering. Agents representing electronic engineering, mechanical engineering, manufacturing, and other disciplines participate in designing electromechanical devices.*

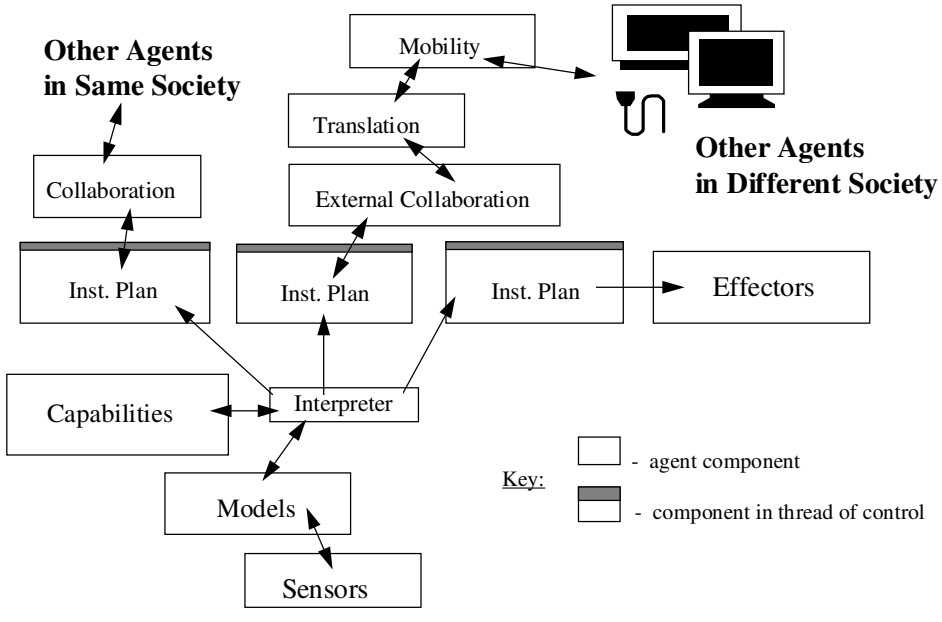

Figure 1: Model of Agent Behavior

3. CONTEXT

Agent- based systems arise out of the following needs:

- Personalized and customized user interfaces that are pro-active in assisting the user
- Adaptable, fault tolerant distributed systems that solve complex problems
- Open systems where components come and go and new components are continually added.
- Migration and load balancing across platforms, throughout a network.
- New metaphors, such as negotiation, for solving distributed, multi- disciplinary problems.

An agent [26] is i) autonomous - acts without human intervention, ii) social - collaborates with other agents via structured messages, iii) reactive - responds to environmental changes, and iv) pro- active - acts to achieve goals. It is the combination of these behaviors that distinguishes an agent from objects, actors [1], and robots. Agent behavior is summarized in Figure 1. Agents have sensors that update their models of the outside world. Agents review theses models and those of themselves to select a capability or plan to address the present situation. Once invoked, each plan executes in its own thread, and several of these may execute concurrently. Agents negotiate with each other; agent collaboration across disciplines may require that semantics can be translated or exchanged.

Three sample capabilities that have been selected and instantiated into a plan are shown in Figure 1. One involves an effector that impacts objects in the outside world; the other two feature collaboration with other agents. One case of collaboration involves another agent within the original society; the other involves an agent in a different society on another computer platform. If the agents are in different societies, they must become mobile and migrate, either virtually or in reality, in order to collaborate.

4. THE LAYERED AGENT ARCHITECTURAL PATTERN

Problem:

How can agent behavior be best organized and structured into software? What software architecture best supports the behavior of agents ?

Forces:

- An agent system is complex and spans several levels of abstraction.
- There are dependencies between neighboring levels, with two way information flow.
- The software architecture must encompass all aspects of agency.
- The architecture must be able to address simple and sophisticated agent behavior.

Solution:

Agents should be decomposed into layers because i) higher level or more sophisticated behavior depends on lower level capabilities, ii) layers only depend on their neighbors, and iii) there is two way information flow between neighboring layers. This approach is a on the Layers architectural pattern [7]. A complete set of layers can be identified from the model of the agent's real world; Fig. 2 structures Fig. 1 into seven layers.

In Figure 2, top down information flow is on the left, while bottom- up is on the right. Bottom-up, an agent's beliefs are based on sensory input. When presented with a problem, an agent reasons to determine what to do. When the agent decides on an action, it can carry it out directly, but an action that involves other agents requires collaboration. Once the approach to collaboration is determined, the actual message is formulated at translation and delivered to distant societies by mobility.

Top- down, distant messages arrive at mobility. An incoming message is translated into the agent's semantics. The collaboration layer determines whether or not the agent should process a message. If the message should be processed, it is passed on to actions. When an action is selected for processing, it is passed to the reasoning layer, if necessary. Once a plan is instantiated and placed in the actions layer, it does not require the services of any lower layers, but it can call on the services of higher ones.

Variations:

The Layered Agent architectural pattern shown in Fig. 2 addresses simple and sophisticated agents. All agents do not have the seven layers shown in Fig. 2. A static agent isn't mobile, so layer 7 is not

present in these agents. A translation layer is only needed for multidisciplinary agent societies (which are presently rare). Many agent societies have one collaboration layer for its agents to share; mobility and collaboration may be combined if an agent society is dispersed over several platforms. If an agent's beliefs about the environment do not change, the agent doesn't need sensors.

Further, aspects of the Layered Agent pattern may be conceptual rather than architectural if, due to performance issues, direct connections between non-adjacent layers are required. For example, reasoning and collaboration may be directly linked if the agent's dominant activity is to service fairly simple requests from other agents.

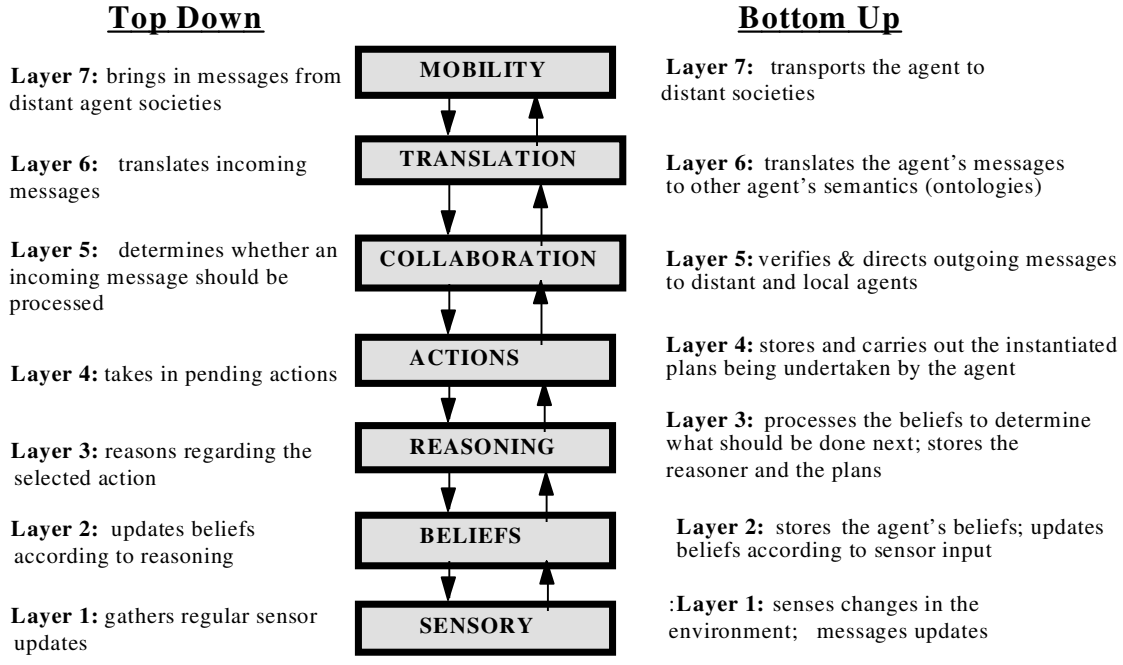

Figure 2: The Layered Agent Architectural Pattern

Known Uses:

There are many layered agent architectures; early ones did not require mobility or translation layers. GRATE [27] features domain, control, and cooperation layers, equivalent to sensory and beliefs, reasoning and action, and collaboration, respectively. TouringMachines [10] consist of perception, action, and control. InterRRaP [20] has four layers: cooperation, plan-based, behaviour-based and world interface.

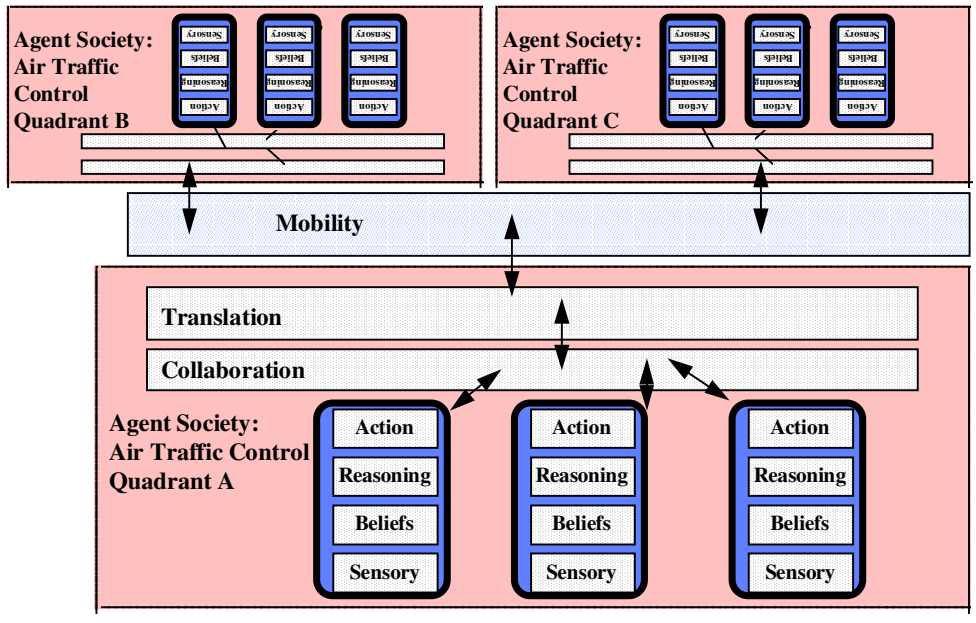

Figure 3: Layered Agents: Centralized Collaboration and Translation and Shared Mobility Layers

Sample Usage:

A sample use of the Layered Agent can be seen in Figure 3. In this, the individual agents each have senses, beliefs, reasoning, and actions. Because the societies are centralized, the agents share a collaboration layer and a translation layer. Three agent societies share a common mobility layer.

5. THE SENSORY, BELIEFS, AND REASONING LAYERS

5.1 CONTEXT AND OVERVIEW

The Sensory and Beliefs layers maintain the agent's models of its environment and itself. Based on these models, the agent determines what to do next in Reasoning. There are four conceptual patterns that have evolved for these three lowest levels of agent capabilities; these patterns differ according to the presence or absence of three factors: a symbolic model; a knowledge based, prescriptive solution; and interaction with a human user. The following table summarizes these four patterns; they are described in a patlet format in the following section.

Sensory, Beliefs, and Reasoning Problem	Symbolic Model	Prescribed Solution	Human Interaction	Solution	Pattern
How can an agent simply react to a stimulus or a request ?	-	-	-	Utilize a stimulus/response type of behavior.	Reactive Agent
How can an agent select a plan to achieve a goal ?	X	X	-	The agent reasons about a symbolic model to select a capability.	Deliberative Agent
How can an agent address problems when no solution is known beforehand ?	X	-	-	Represent the problem's constraints, and let the agent opportunistically solve the problem.	Opportunistic Agent
How can an agent adapt to the needs of a human user ?	-	-	X	Provide the agent with parametric user models and sensors to monitor the user's actions.	Interface Agent

5.2 THE REACTIVE AGENT

Problem

How can an agent react to an environmental stimulus or a request from another agent when there is no symbolic representation and no known solution ?

Forces

- An agent needs to be able to respond to a stimulus or a request.
- There may not be a symbolic representation for an application.
- An application may not have a knowledge based, prescriptive solution.

Solution

A Reactive Agent does not have any internal symbolic models of their environment; it acts using a stimulus/ response type of behavior. It gathers sensory input, but its Belief and Reasoning layers are reduced to a set of situated action rules. A single Reactive Agent is not proactive, but a society of these agents can exhibit such behavior. A Reactive Agent is known as a weak agent.

Known Uses

Reactive theory was originated by Brooks [6] and Agre and Chapman [2]; reactive agents have been widely used [22] to simulate the behavior of ant societies and to utilize such societies for search and optimization [9].

5.3 THE DELIBERATIVE AGENT

Problem

How can an agent select a capability to proactively achieve a goal within a given problem context ?

Forces

- An agent should be capable of intelligent behavior, selecting a plan to achieve a goal.
- For some applications, a symbolic representation or model of the environment can be specified.
- Some problems have a knowledge based solution that can be identified by experts.

Solution

A Deliberative Agent possesses an internal symbolic reasoning model of their environment and themselves within their Beliefs and Reasoning layers. They select a plan or capability that can achieve their goal in the context of the present situation. A Deliberative Agent is a strong agent, and a sample use involves a society of agents with knowledge of particular business processes.

Known Uses

Deliberative Agents were originated by Cohen [8] and Georgeff [12], and they have been widely used by Jennings [15] and others [22].

5.4 THE OPPORTUNISTIC AGENT

Problem

How can an agent opportunistically address problems, identifying an approach that is not known appriori ?

Forces

- A problem can have a symbolic representation but not have a knowledge based, prescriptive solution.
- For these applications, only constraints may be known; these indicate what can not be done.
- An agent needs to be able to avoid known constraints but still move toward a solution.

Solution

An Opportunistic Agent does not attempt to have prescriptive plans to address a problem. Rather, their Beliefs consist of constraints found in the problem, and their Reasoning or capabilities accomplish constraint propagation and satisfaction. Problems with a symbolic representation but with no known appriori, prescriptive solution can be solved this way.

Known Uses

Fox [4, 21, 23] has pioneered this approach and used it successfully in distributed scheduling and resource allocation; these problems typically have no knowledge based approach.

5.5 THE INTERFACE AGENT

Problem

How can an agent adapt to the needs of a human user ?

Forces

- Some agents work directly with a human user, assisting them in using an application or in finding information or services.
- The needs of human users are variable, but there are certain categories of users and established patterns of user behavior.

Solution

An Interface Agent collaborates with a human computer user. Typically, only one agent is found, although a full agent society may be used. This kind of agent observes the user and adapts to their needs by identifying what kind of user they are and their patterns of computer useage. An Interface Agent's beliefs are typically parametric user models, and their sensors monitor the user's actions.

Known Uses

Maes [19] has led the development of Interface Agents, also called Personal Assistants [22].

6. THE ACTION LAYER

6.1 CONTEXT AND OVERVIEW

The Action layer carries out the plan selected by the Reasoning layer. There is a need for the layer to be able to schedule and prioritize actions. It makes use of the conceptual and design patterns summarized below:

Action Problem:	Solution:	Pattern:
How can an agent commit to behavior ?	An instantiated plan is an Intention that executes in its own thread of control.	Intention
How can a plan or request be encapsulated as an object ?	Implement a plan interface for a high level operation with different subclasses.	Plan as Command
How can different plans, intentions and requests be instantiated at runtime ?	Provide an interface for creating families of related objects without specifying their concrete classes. Or, let subclasses determine which class to instantiate.	Plan and Intention Factory
How do you manage different threads of control for agent actions and migrations?	Decouple method execution from method invocation to simplify synchronized access to a shared resource by methods invoked in different threads of control.	Adaptable Active Object
How can priority handling and other forms of behavior be added to an intention dynamically ?	Decorate or add behavior to the run () method of the intention thread, where a plan executes.	Prioritizer
How can messages from other agents be passed to the agent's reasoning capability ?	Decouple reasoning execution from invocation to simplify synchronized access to the agent's shared reasoning resource.	Message Forwarder
How can two separate threads, an intention and a concurrent server, communicate asynchronously ?	Provide a placeholder for a result. Provide an observable relationship between the server and the placeholder.	Future Observer

6.2 THE INTENTION

Problem

How can an agent commit to performing reactive and proactive behavior ?

Forces

- An agent needs to be able to commit to proactive and reactive behavior.
- Behavior executes with the beliefs that the agent had when it (the behavior) was initiated.
- An agent may have many activities or plans executing concurrently.
- An agent's plan impacts the environment through the effectors; it calls on collaboration when it needs to involve other agents.

Solution

An Intention represents the commitment of an agent to being in a state where it believes it is about to actually perform a set of actions [9]. An instantiated plan is an Intention that executes in its own thread of control; it executes until completion, unless it is suspended awaiting a reply. A plan's goals are stated in invocation conditions; additional criteria, such as environmental situations or stimuli, are in context conditions. Conditions and plans reside in the Reasoning layer (Figure 4). If the conditions are satisfied, the plan is instantiated and executed by an Intention in the Actions layer. All variables and expressions in the plan are evaluated, based on the agent's beliefs, at the time of instantiation, when the agent commits to performing the plan. An Intention can be specialized to a CollaborationIntention and a ReactionIntention (Figures 4 and 5). Once an Intention is created, it does not require the services of any of the lower layers; collaboration can involve higher layers.

Known Uses

Intentions were first introduced by Georgeff and Lansky [12], as part of their Belief- Desires-Intentions agent architecture. Intentions provide the proactive and reactive behavior of many strong and weak agent systems, including [8] and [15].

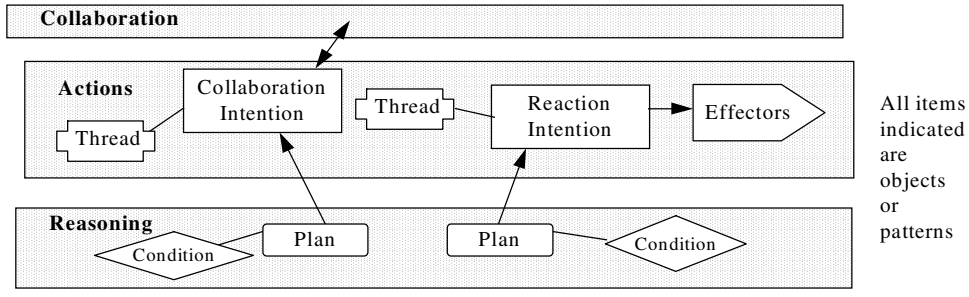

Figure 4: Intentions in the Action Layer, Plans and Conditions in the Reasoning Layer

6.3 PLAN AS COMMAND

Problem

How can a plan be encapsulated as an object ?

Forces

- Each Intention has a plan to execute. They have a wide range and are known only at run time.
- A plan specifies primitive actions, executed directly by the effectors or the Collaboration layer interface.
- There is a need to define a structure for plans that provides high level operations based on primitive ones.

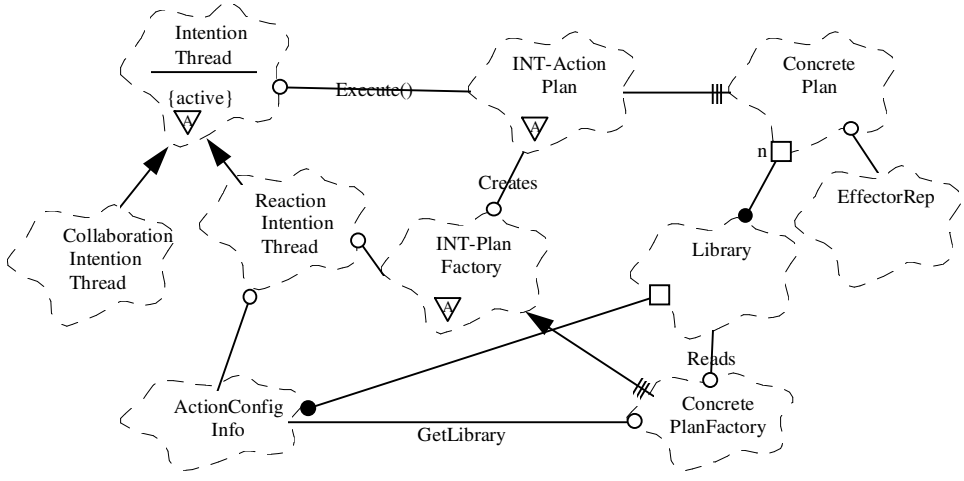

Figure 5: The Intention, Plan as Command, and Plan and Intention Factory Patterns

Solution

The Plan as Command pattern [11] solves this problem, as shown in Figure 5 in three objects/ interfaces: INT- ActionPlan, ConcretePlan, and EffectorRep. (NOTE: This figure and all following object diagrams utilize Booch notation.) Each ConcretePlan is a command object which implements

the ActionPlan interface that declares the high level operation Execute(). The receiver of this command is a ConcretePlan object which is instantiated at runtime. Each ConcretePlan uses primitive methods of EffectorRep and the Collaboration interface (not shown).

Known Uses

The Plan as Command pattern was found to be a recurring problem and solution in the Java Application Framework for Intelligent and Mobile Agents (JAFIMA) Project.

6.4 PLAN AND INTENTION FACTORY

Problem

How can different plans and intentions be instantiated at run time ?

Forces

- A plan is instantiated for every new Intention, and they must all utilize the same interface for creation.
- The type of intention (Collaboration or Reaction) will depend on the action plan to be executed; therefore, the type of intention thread to be instantiated can not be anticipated before run time.
- There is a need to delegate the responsibility of instantiating intention objects.
- New subclasses of intentions may become necessary.

Solution

The Abstract Factory and Factory Method patterns [11] are used together to form the Plan and Intention Factory, as shown in Figure 5. The PlanFactory is abstract; it provides the interface to create the ActionPlan objects. The ConcretePlanFactory provides the implementation to create the ConcretePlans. For this it uses the Library which stores several ActionPlan classes and instantiates the requested one. This plan object is then used by the IntentionThread. Per the Factory Method pattern, ReactionIntention or CollaborationIntention subclasses will be instantiated depending on the ActionPlan type. Both of these define the virtual methods of the IntentionThread class. These methods are used by the IntentionThread class for creating the respective intention thread objects, letting the subclasses determine how an object is to be instantiated.

6.5 THE PRIORITIZER

Problem

How can priority handling and other forms of behavior be added to an intention dynamically ?

Forces

- There are two main Intention subclasses: Reaction and Collaboration. Additional refinement is needed, especially for priority handling.
- Further subclassification will result in duplication, as both Reaction and Collaboration Intentions can feature the same priority handling.
- Priority handling should be attached to an object, and not a class, because the type of IntentionThread is not known before run time.

Solution

Additional responsibilities can be attached to an Intention dynamically using Decorators [11]. The Prioritizer pattern can be used to decorate the run() method of the IntentionThread, where action plans are executed. The run() method is declared in Runnable interface and is called whenever a thread is started. Additional priority handling can be added dynamically to it by using the decorator object, the ThreadControl class that encapsulates the IntentionThread. Both the ThreadControl and the IntentionThread classes conform to the Runnable interface, so the instance of ThreadControl class can be used transparently in place of IntentionThread. The subclass of ThreadControl, the Controller class, provides the concrete decorator.

Known Uses

The Prioritizer pattern was utilized in the JAFIMA Project.

6.6 FUTURE OBSERVER

Problem

How can two separate threads, an intention and a concurrent server, communicate asynchronously ? How can the dependent intention be notified of the server's response ?

Forces

- In Java, threads can not return results directly as Runnable.run() has a void return type.
- In an agent, two separate threads, one of an intention and another of a concurrent server, have to be able to communicate asynchronously when a result is returned from the server thread.
- There can potentially be many intentions executing, and only some of these will be dealing with a given concurrent server.

Solution

In the Future pattern [16], an instance of Future is used as a placeholder for a future value. In the Observer pattern [11], a one to many dependency is defined so that dependents can be notified when the observable changes. The Future Observer pattern combines these to solve the problem stated above. The Client will execute an asynchronous operation, DoOperation(), which instantiates a Future object and returns its reference. The Client will message the Future object's read() which will block thread execution if the Future is not in its updated state. Later on, the concurrently executing CoexistingServer updates the state of the Future object. Each Future object is an observable for the corresponding observer Clients who register themselves with the related Futures. When a Future is updated by the CoexistingServer it notifies these observers. This notification will execute the update method of the Client, and in this method the blocked (or suspended) Client thread is resumed.

Known Uses

The Future Observer pattern was employed in the JAFIMA Project.

6.7 ADAPTABLE ACTIVE OBJECT

Problem

How do you manage different threads of control for agent actions ? How can the agent's actions conform to different environments ?

Forces

- Agent intentions act concurrently in different threads of control.
- An object in the environment may need to be affected or impacted by the agent in a sequential manner.
- Agents may act in various environments, with different effectors.
- The Active Object pattern uses MethodObjects, but it is not practical to represent each method as a separate class and instantiate it at runtime because of the variability in effectors.

Solution

The Active Object pattern [18] decouples method exectuion from method invocation in order to simplify synchronised access to a shared resource. ClientInterface, Scheduler and ActivationQueue form the Active Object pattern, along with Method Object; however, new Method Object classes would be necessary for each method in each environment. The solution to this problem is provided by the Adapter pattern [11] and the class ConcreteAdapter. The user has to provide the ConcreteAdapter which marshalls the method call when a method is invoked and later on demarshalls the method object when it is dispatched.

Known Uses

The Adaptable Active Object pattern was found to be a recurring problem and solution in the Java Application Framework for Intelligent and Mobile Agents (JAFIMA) Project.

6.8 THE MESSAGE FORWARDER

Problem

How can messages from other agents be passed through the Action layer to the agent's reasoning capability ?

Forces

- Messages arrive from other agents concurrently.
- The agent's Reasoning layer is sequential.

Solution

The Message Forwarder, based on the Active Object [18] decouples sequential reasoning execution from concurrent requests to simplify synchronized access to the agent's shared resource.

Known Uses

The Message Forwarder pattern was found to be a recurring problem and solution in the JAFIMA Project.

7. THE COLLABORATION LAYER

7.1 CONTEXT AND OVERVIEW

In the Collaboration layer, the agent determines its approach to cooperating or working with other agents. Conceptual, architectural, and design patterns are utilized for messaging (Conversation), centralization (Facilitator), decentralization (Agent Proxy), and social policies (Protocol, Emergent Society); they are summarized below:

Collaboration Problem:	Solution:	Pattern:
How can messaging between agents occur in sequences ?	Agent messaging can occur within a context established by previous messages.	Conversation
How can agents collaborate without direct knowledge of each other ?	Encapsulate agent interaction in a Facilitator that coordinates agents within a given society.	Facilitator
How can agents collaborate directly with one another ?	Provide a Proxy to control access to the agent and to provide distinct interfaces. Store and retrieve conversations.	Agent Proxy
How can agent collaboration be prescribed ?	Establish conversation policies that explicitly characterize communication sequences.	Protocol
How can agents cooperate to achieve goals when there is no established protocol ?	Though stimulus/response behavior, each agent can stimulate its neighbors. Complex patterns of behavior emerge when viewed globally..	Emergent Society

7.2 THE CONVERSATION

Problem

How can structured messaging between agents occur in sequences rather than in isolated acts ?

Forces

- Successive messages between agents are often related.
- Endless loops of messages between agents need to be avoided.

Solution

A Conversation [5] is a sequence of messages between two agents, taking place over a period of time. There are termination conditions for any given occurrence, and Conversations may give rise to other Conversations. In some agent societies, messages between agents may occur only within the context of conversations; isolated messages are not supported.

Known Uses

COOL [4] and AgenTalk [22] support Conversations between agents, as does KAoS [5].

7.3 CENTRALIZED COLLABORATION: THE FACILITATOR

Problem

How is an agent able to freely collaborate with other agents without direct knowledge of their existence?

Forces

- Each agent may not have knowledge of every other agent
- Proliferating interconnections and dependencies increase complexity, complicate maintenance, and reduce reusability

Solution

Each Mediator [11] is associated with a multitude of Colleagues, objects that rely on it for all communication. The Facilitator is based on the Mediator, and it provides a gateway or clearinghouse for agent collaboration [5]. With a Facilitator, agents do not have to have direct knowledge of one another for collaboration, and agents within the same society share a single Collaboration layer.

Known Uses

ARCHON [15], PACT [25], and other agent applications have utilized Facilitators, refering to this approach as a federated agent architecture [22].

7.4 DECENTRALIZED COLLABORATION: THE AGENT PROXY

Problem:

How can agents collaborate directly with one another?

Forces:

- An agent may not have a Facilitator to represent it. Then, each agent must communicate directly with other agents, support different interfaces, and maintain collaboration knowledge.
- Agents collaborate with each other via structured messages; there are many agent dialects.
- Bottlenecks encountered in a centralized architecture need to be avoided.
- An agent must be able to recover Conversations that it is involved in.

Solution:

A Proxy [11] controls access to the Real Subject; it can also provide a distinct interface. Each Agent Proxy class (Figure 6) would subscribe to a certain interface. An agent must be able to determine its behavior based upon the state of the conversation it is involved in. One agent may be engaged in several conversations simultaneously, requiring context switching. The Memento pattern [11] externalizes an object's state so that the state can be restored later. Agent Proxies that support conversations must store and recover their state, delegating this to a Memento.

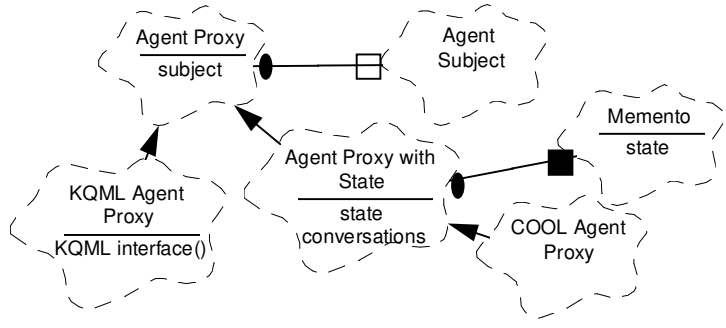

Figure 6: The Agent Proxy Pattern for Decentralized Collaboration

Known Uses

The Agent Proxy pattern was found to be a recurring problem and solution in the Java Application Framework for Intelligent and Mobile Agents (JAFIMA) Project. Proxy Agents also appear in KAoS [5].

7.5 PROTOCOL

Problem

How can agent collaborative behavior be prescribed to follow certain policies ?

Forces

- Agents need to be able to follow certain conventions or policies for collaboration.

Solution

Conversation policies [5] or Protocols prescriptively encode regularities that characterize communication sequences between users of a language. Agent Protocols explicitly define what sequences of which messages are permissible between a given set of participating agents.

Known Uses

COOL [4] prescribes a particular form of agent negotiation. KAoS [5] and AgenTalk [22] stipulate several protocols or conversation policies, including contract net, inform, offer, and request [5]. In the contract net protocol, one agent asks for bids for tasks it needs performed, and other agents respond, if they are available to do the work. If a bid meets the originating agent's criteria, it can award the work to the successful bidder.

7.6 THE EMERGENT SOCIETY

Problem

How can agents collaborate without known protocols ? How can Reactive Agents collaborate ?

Forces

- There may not be known agent protocols for a given application.
- Reactive Agents need to be able to collaborate and carry out proactive behavior together.
- Reactive Agents simply react to stimuli and are not capable of any knowledge based behavior.

Solution

Each individual agent, even a Reactive Agent, can, through their own actions, provide a stimulus to a neighboring agent. As each individual agent reacts to stimuli provided by their neighbors, the net result is the Emergent Society. Complex patterns of behavior can emerge from these interactions when the agent society is viewed globally [22]. No model exists for this behavior, although economic and game theory have been applied successfully. Reactive Agents and agents from Emergent Societies have reduced Collaboration layers; they merely provide stimuli to neighboring agents.

Known Uses

All Reactive Agent systems [9] rely on the Emergent Society for collaboration [22].

8. THE MOBILITY LAYER

8.1 CONTEXT AND OVERVIEW

The Mobility layer must support real and virtual migration. It consists of a region shared across several agents and agent societies, and a region that belongs to an individual agent. It is made up of the following architectural, conceptual, and design patterns:.

Mobility Problem:	Solution:	Pattern:
How can an agent become resident in a distant society ?	Replicate the agent, providing sensors and effectors for the new environment.	Clone
How can an agent be cloned in a distant society ?	Define operations for cloning in the destination society without changing the agent class. Separate the construction of the agent from its representation.	Remote Configurator
How is an agent able to gain access to resources and other agents outside its society transparently ?	Location transparency is provided by a Broker. Proxies must be employed for the client and server to be able to respond to the interface of the Broker.	Broker
How can an agent migrate virtually or in reality, dynamically ?	Provide a Thread Manager and a Handler Creator, and allow the subclasses for virtual and actual migration to address thread instantiation.	Migration Thread Factory

8.2 THE CLONE

Problem

How can an agent relocate itself and become resident in distant societies ?

Forces

- An agent must be able to bring its capabilities, facilities, and state with it to a new society.
- The agent must be able to travel to a remote location and interact, negotiate, and exchange information in the new society.

Solution

Make a copy or clone of the original agent, and place the new agent in the distant society. The clone must have all of the capabilities and facilities of the original agent, along with any state information.

Known Uses

The original use of agent self replication was cooperating mobile WAVE agents [3]. More recent approaches that utilize cloning include IBM Aglets [13], and the Agent Transfer Protocol (ATP) [14]. An aglet is a Java object that can move from one host on the Internet to another. When the aglet moves, it takes along its program code as well as its state (data). Bradshaw [5] refers to agent cloning as teleportation.

8.3 THE REMOTE CONFIGURATOR

Problem

How can an agent be appropriately configured for various destination societies ?

Forces

- For actual migration, an agent has to be cloned in the destination society. Configuration details are needed for cloning, such as the plan library and the beliefs.
- The configuration details and their format depend on the given society's requirements. Thus each agent has to support various kinds of configuration access operations.
- There is a need to represent the configuration accessing functions separately from the agent structure; otherwise each agent has to support many distinct and unrelated operations in object structure.
- Each agent has a similar object structure as all of them are created by the same framework

Solution

Figure 7 shows how the Visitor pattern [11] has been utilized to design the Remote Configurator. The ActualMigration handler transfers the Visitor from a distant society to the migrating agent. The ClientProxy in the Mobility layer instantiates the Visitor object. This Visitor object is passed to the corresponding layers, such as Reasoner, by calling Accept(). As shown, the Reasoner object in turn calls back the Visitor object's Visit method and passes its reference to the Visitor object. As the object structure of the layer of an agent is fixed, the Visitor can gather the configuration information by using the public interface methods, such as GetPlan(), of the Reasoner. Thus there is no need to define separate methods for transferring the configuration details to another society. Moreover, services can be added by adding new Visitor subclasses, and no change is needed in the agent structure.

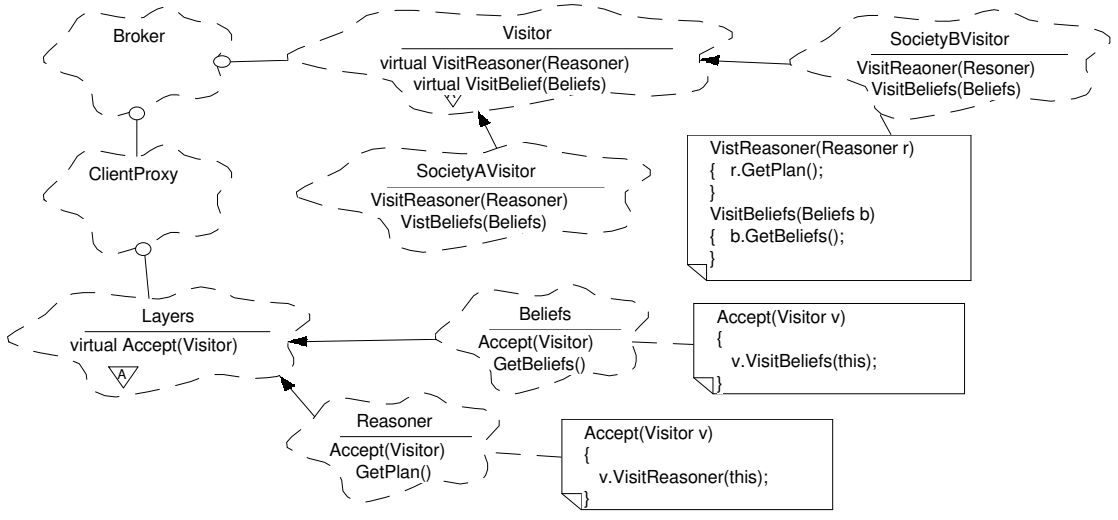

Figure 7: The Remote Configurator in the Mobility Layer

Known Uses

The Remote Configurator pattern was used in the JAFIMA Project.

8.4 THE BROKER

Problem:

How is an agent able to gain access to resources and other agents outside its society without actually migrating ?

Forces:

- Agents must be able to access each other and other resources across platforms and societies without having to actually migrate.
- Making every agent responsible for access, security and interactions for a society leads to N- to- N connections and redundancy.

Solution:

The Broker pattern [7] provides for location transparency for objects that wish to be clients and servers of one another. With this, the agent (or its Agent Proxy) can become a virtual member of

open societies managed by the Brokers. Bridges between societies are also supported. Agents who wish to be clients and servers for one another must employ a Broker who is responsible for locating a server once a client has requested its services. Both the client and the server must register with the Broker. The Broker pattern provides virtual agent migration.

Known Uses:

Bradshaw refers to a Broker as a Matchmaker [5].

8.5 MIGRATION THREAD FACTORY

Problem

How can an agent migrate virtually or in reality, dynamically ?

Forces

- An agent can request a service from another society (virtual migration) or it can migrate physically.
- The type of the request is only known at runtime, and the behaviour required for each type is very different.
- Requests occur concurrently, and services should be concurrent.
- There is a need to dynamically create the handler object according to the incoming request.
- New types of migration services may need to be added.

Solution

Figure 8 shows the design of the Migration Thread Factory [11], where the Broker has a ThreadManager and a HandlerCreator. The abstract class Handler declares the factory method MakeHandler in its interface and uses this factory method in the Create(Message). The concrete definition of this factory method is given by the subclasses VirtualMigration and ActualMigration. The HandlerCreator selects the appropriate subclass according to the client request for migration and instantiates it. It then calls Create() on this object which then composes itself and creates the object by using MakeHandler().

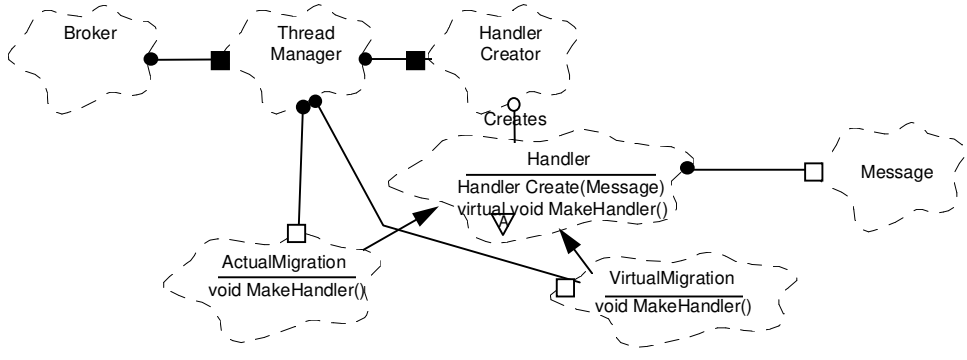

Figure 8: The Migration Thread Factory

Known Uses

The Migration Thread Factory pattern was found to be a recurring problem and solution in the Java Application Framework for Intelligent and Mobile Agents (JAFIMA) Project.

9. PATTERNS FOR AGENT CONFIGURATION AND INTEGRATION

9.1 CONTEXT AND OVERVIEW

The process of creating and configuring an agent consists of creating the various layers and then integrating them. The design for creating the object structure of individual layers and integrating them uses the following patterns: Agent Builder and Layer Linker.

Configuration/ Integration Problem:	**Solution:**	**Pattern:**
How can the construction of the agent be separated from its representation ?	Delegate layer construction to a Builder and a Director.	Agent Builder

How can a simple interface between the layers be provided ?	Implement a unified interface for each layer. Provide Facades for each layer interface.	Layer Linker

9.2 THE AGENT BUILDER

Problem

How can the construction of the complex object structure of the agent be separated from its representation so that the same construction process can create different representations?

Forces

- Each agent has fundamentally the same structure.
- The creation process of the agent should be isolated so that the same process can be used to generate different object structures.

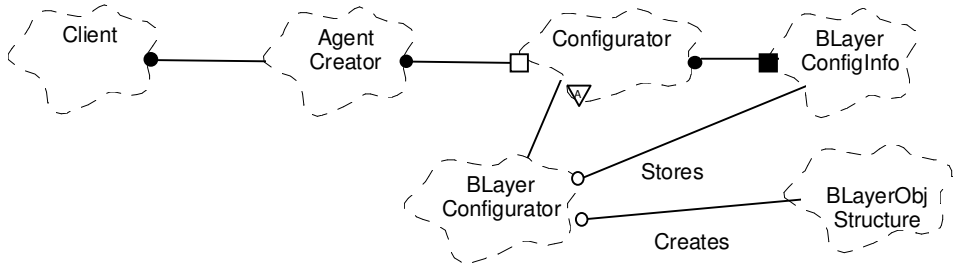

Figure 9: The Agent Builder for Creating the Object Structure of Each Agent Layer

Solution

The AgentCreator collects the user configuration details and passes it to the Creator object. According to the Builder pattern [11], Creator is a director object (Figure 9). It stores the process used to create an agent. The AgentCreator instantiates the Creator with seven Configurator objects, and the Configurator objects are the builder objects which actually create the object structure of each individual layer based on the information in the Configuration. The Configurator declares a virtual method Configure. The concrete definition of this method is provided by the layer specific subclasses.

Known Uses

The Agent Builder pattern was found to be a recurring problem and solution in the JAFIMA Project

9.3 LAYER LINKER

Problem :

How can the various individual layers of the agent be integrated together ?

Forces

- It is necessary to provide an interface to each layer but also to decouple the layers as much as possible.

Solution

Each Configurator creates the Facade object and other objects which form the structure configured by an agent application developer. Use of the Facade pattern [11] provides both a simple interface and decoupling between the layers. The Facade object implements the unified interface for the layer, promoting layer independence and portability. For example, layers A and C are neighbouring layers to layer B. Then, INT-CB Interface declares the interface from layer C to layer B, and, correspondingly, INT-AB Interface declares the interface from layer A to layer B. The Configurator registers them in the configuration repository object BlayerConfigInfo. These repositories are used to get the Facade object reference during the integration phase executed by the Creator object's integrate().

Known Uses

The Layer Linker pattern was utilized in the JAFIMA Project.

10. REFERENCES

1. Agha, G., A Model of Concurrent Computation in Distributed Systems. 1986: MIT Press.
2. Agre, P. E., and D. Chapman, "Pengi: An Implementation of a Theory of Activity," Proceedings of the 6th National Conference of Artificial Intelligence, 1987.

3. Al- Jabir, S., Sapaty, P. S., and Underhill, M., “Integration of Heterogeneous Databases Using WAVE Cooperative Agents,” Proceedings of the First International Conference on the Practical Application of Multi Agent Systems, London, 1996.
4. Barbuceanu, M. and M.S. Fox, "COOL: A Language for Describing Coordination in Multi-Agent Systems", First International Conference on Multi-Agent Systems, 1995.
5. Bradshaw, J. M., S. Dutfield, P. Benoit, J. D. Woolley, “KAoS: Toward an Industrial- Strength Open Distributed Agent Architecture,” J.M. Bradshaw (Ed.), *Software Agents*, AAAI/ MIT Press, 1997.
6. Brooks, R.A., "A Robust Layered Control System for Mobile Robot", IEEE Journal of Robotics and Automation, 1986. RA-2(1).
7. Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal*, Pattern-Oriented Software Architecture: A System of Patterns*. Wiley and Sons. 1996:
8. Cohen, P. R., and Levesque, H. J., “Intention is Choice with Commitment,” Artificial Intelligence, 42 (3), 1990.
9. Ferber, J., “Simulating with Reactive Agents,” in Many Agent Simulation and Artificial Life,” E. Hillebrand and J. Stender, Editors, Amsterdam, IOS Press, pp. 8 - 28, 1994.
10. Ferguson, I.A., "Towards an Architecture for Adaptive, Rational, Mobile Agents", Proceedings of the Third European Workshop on Modelling Autonomous Agents and Multi-Agent Worlds (MAAMAW-91)*,* 1991.
11. Gamma, E.R., R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. 1994: Addison-Wesley.
12. Georgeff, M.P. and A.L. Lansky, "Reactive Reasoning and Planning", Proceedings of the Sixth National Conference on Artificial Intelligence, Seattle, WA, 1993.
13. IBM: Aglets: Programming Mobile Agents in Java, http://www.trl.ibm.co.jp/aglets, 1997.
14. IBM: Agent Transfer Protocol, http://www.trl.ibm.co.jp/aglets/atp/atp.html, 1997.
15. Jennings, N.R., P. Faratin, M. Johnson, P. O'Brien, and M. Wiegand, "Using Intelligent Agents to Manage Business Processes", First International Conference on the Practical Application of Intelligent Agents and Multiagent Technology, London, 1996.
16. Kafura, D.G., “A Polymorphic Future and First Class Function type for Concurrent Object Oriented Programming”, Journal of Object Oriented Systems.
17. Kendall, E.A., M.T. Malkoun, and C.H. Jiang, "A Methodology for Developing Agent Based Systems for Enterprise Integration", EI '95, IFIP TC5 SIG Working Conference on Models and Methodologies for Enterprise Integration, Heron Island, Australia, 1995.
18. Lavender, R.G. and D.C. Schmidt, "Active Object: an Object Behavioral Pattern for Concurrent Programming", *Pattern Languages of Programming*, Illinois, 1995.
19. Maes, P., “Agents that Reduce Work and Information Overload,” Communications of the ACM, Vol. 37, No. 7, pp. 31 - 40, 1994.
20. Muller, J.P., M. Pischel, and M. Thiel, "Modelling interacting agents in dynamic environments", Proceedings of the Eleventh European Conference on Artificial Intelligence (ECAI-94)*,* Amsterdam, 1994.
21. Nwana, H. S., L. Lee, and N. R. Jennings, “Coordination in Multi- Agent Systems,” in *Software Agents and Soft Computing, Towards Enhancing Machine Intelligence*, H. S. Nwana and N. Azarmi, Editors, Springer, 1997.
22. Nwana, H. S. and D. T. Ndumu, “An Introduction to Agent Technology,” in *Software Agents and Soft Computing, Towards Enhancing Machine Intelligence*, H. S. Nwana and N. Azarmi, Editors, Springer, 1997.
23. Sathi, A., and M. Fox, “Constraint- Directed Negotiation of Resource Allocations,” in *Distributed Artificial Intelligence 2*, L. Gasser and M. Huhns, Editors, Morgan Kaufmann, 1989.
24. Schmidt, D.C., "The ACE Object-Oriented Encapsulation of Light Weight Concurrency Mechanisms", 1995.

25. Tenenbaum, J.M., J.C. Weber, and T.R. Gruber, “Enterprise Integration: Lessons from SHADE and PACT, in Enterprise Integration Modeling,” Proceedings of the First International Conference, C.J. Petrie, Editor. 1992, MIT Press.
26. Ueno, I., S. Yoshida, K. Kuwabara, “ A Multi Agent Approach to Service Integration”, Proceedings of the Second International Conference on the Practical Application of Intelligent Agents and Multi Agent Technology, London, 1997.
27. Wooldridge, M.J. and N.R. Jennings, "Agent Theories, Architectures and Languages", ECAI-94 Workshop on Agent Theories, Architectures, and Languages, Amsterdam, ed. J.G. Carbonell and J. Siekmann, Springer - Verlag, 1994.