Agent Architectures and Interaction Protocols for Corporate Memory Management Systems

Federico Bergenti, Agostino Poggi and Giovanni Rimassa[1]

Abstract. This paper discusses what are the most suitable agent architectures and interaction protocols to implement a multi-agent system for the management of a corporate memory helping the users into two main tasks: the insertion of new documents and the retrieval of information. Moreover, the paper presents a software framework called JADE (Java Agent Development Environment), and shows how it can be used to develop the agent architectures and interaction protocols we need to realise such a multi-agent system. These guidelines should be token in account during the realisation of such a system we are developing together some other research centres inside the European Commission IST project "CoMMA - Corporate Memory Management through Agents".

1 INTRODUCTION

A corporate memory is defined in [21] as an "explicit, disembodied, persistent representation of knowledge and information in an organisation, in order to facilitate its access and reuse by members of the organisation, for their tasks".

In order to realise such kind of system, some recent works shown that organisations can take advantages of internet and intranet technologies and of agents. Internet and intranet technologies simplify the diffusion of knowledge. In fact, the Web can serve as a basis to distribute information in a uniform way independently of the way the information is stored [6,9,21]. Agents can help the user supporting retrieval of the relevant information from the corporate memory and adapting the interaction with the system to the user's preferences [2].

In this paper, we discusses what are the most suitable agent architectures and interaction protocols to implement a multi-agent system for the management of a corporate memory we are developing together some other research centres inside the European Community IST project "CoMMA - Corporate Memory Management through Agents" [5]. Moreover, the paper presents a software framework called JADE (Java Agent Development Environment), and shows how it can be used to develop the agent architectures and interaction protocols we need to realise such a multi-agent system.

[1] Dipartimento di Ingegneria dell'Informazione, University of Parma, Parco Area delle Scienze, 181A, 43100, Parma, Italy, *email:* {bergenti, poggi, rimassa)@CE.UniPR.IT

2 AGENT TYPES FOR CORPORATE MEMORY MANAGEMENT

The corporate memory management multi-agent system we are developing should help the user in three main tasks: the XML annotation and insertion of new documents and the search of documents, and should autonomously push her/him information about new interesting documents.

To perform these tasks, the system needs five categories of agents:

- a set of Interface Agent (IAs), one for each user, operating as personal assistant guiding its user both in the search of documents in the corporate memory and in the insertion of new documents in the corporate memory;
- a set of User Profile Agents (UPAs) managing user profiles and driving information pushing;
- a set of Document Ontology Agent (DOAs) managing document ontologies used to annotate documents;
- a set of Search Agent (SAs) searching documents in the corporate memory;
- a set of Archivist Agents (AAs) storing documents in the corporate memory.

Other kinds of agents as, for example, Directory Facilitators, Mediators and Resources Managers might be present.

The XML annotation and insertion of new documents can be described as follows. At the beginning of the XML annotation the IA negotiates with the DOAs which of them will help it to guide the user in the annotation. During the annotation of a document, the IA changes information with the user's UPA receiving information to prevent user's actions and sending information about what the user really do. Finally, when the annotation is finished and the document is ready to be stored the IA negotiates with the AAs which of them will store the document. Note that each user has a fixed UPA because the IA negotiates its help after user's registration.

The search of documents can be described as follows. At the IA negotiates with the DOAs which of them will help it to guide the user to build the query. During the generation of the query, the IA changes information with the user's UPA receiving information to prevent user's action and sending information about what the user really do. Finally, when the query is ready to be executed the IA negotiates with the SAs which of them will search the document. The SA agent tries to directly execute the user's query, if the query has not results, the SA negotiates with the DOAs which of them will help it to refine the query. Moreover, the SA can use the partial results of the query execution to build new queries (e.g., if the query is "find all the

documents whose author is Mayer and whose topic is agent systems" and the SA finds only a document about it where Mayer is one of the authors, then this SA refines the query, adding the other authors of the found document as possible alternative to Mayer, executes the new query adding its results to first found document). After that the SA gives the results to the IA that, helped by user's UPA, orders the results and presents them to the user.

If the corporate memory is distributed, the IA negotiates with the SAs which of them will search the document, but the chosen SA builds a team of SAs (one for each memory node), searches in a node of the corporate memory delegating the search in the other nodes to the other team members. At the end, it collects their results and then sends the results to the IA. In this case, a SA might work independently from the other SAs involved in the search; however, they may co-operate to improve the search. They may co-operate in the generation of a common global query. For example, if all the team members start the search with the user query, but none of them succeeds in finding enough information, they can refine the query using both the DOAs help and the information of the partial results, and then they can propagate the new query to the other team members (e.g., if the query is "find all the documents whose author is Mayer and whose topic is agent systems" and only an SA finds a document about it where Mayer is one of the authors, then this SA refines the query adding the other authors of the found document as possible alternative to Mayer, executes the new query and informs the other team members about this new opportunity).

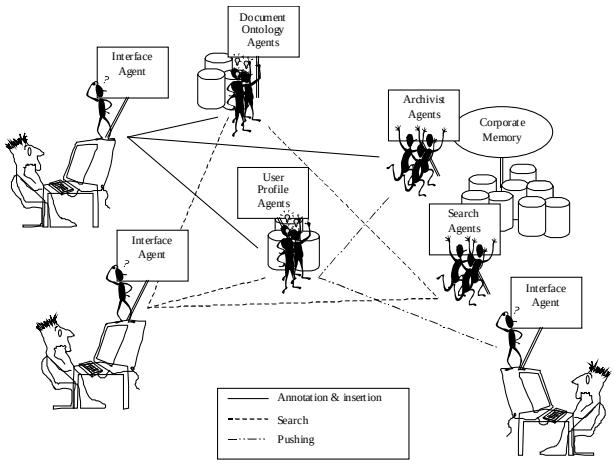

Figure 1. Graphical description of the interactions among the agents during the annotation and insertion, search and pushing tasks.

The pushing of documents is driven by:

- unsatisfied user's requests (i.e., a user is not satisfied by the result of a query),
- a persistent user's requests (i.e., a user subscribes herself to a specific interest), or
- a foreseeable interest (i.e., the UPA reasoning on its users profiles determinates that some of the new information might be of great interest to an its user).

This task can be described as follows. When an AA stores a new document in the corporate memory, it informs the UPAs. These agents check the unsatisfied and persistent users requests and, if the document do not satisfy such requests for some users, the UPAs reason on their profile to check if this document might be of interest for their users. When the IA informs its UPA that the user enters the system, the UPA sends to the IA the information about the new documents that might be of interest to its user.

3 AGENT ARCHITECTURES FOR CORPORATE MEMORY MANAGEMENT

The first step to realise such a multi-agent system is to choose some appropriate agent architectures for the different types of agent the management of our corporate memory needs. This choice is done through an analysis of the agent types involved in the management of the corporate memory on the basis of the weak notion of agent [24], and through an analysis of the most known agent architecture models.

3.1 Agent types characterisation

The weak notion of agent characterises an agent through the following properties:

- autonomy, an agent can operate without the direct intervention of humans or other agents, and has a complete control over its action and internal state;
- reactivity, an agent can perceive its environment and respond in a timely fashion according to changes that occur within that environment;
- pro-activity, an agent does not simply act in response to its environment, but it is able to exhibit opportunistic, goal directed behaviour by taking the initiative where appropriate;
- social ability, an agent can interact with other agents and, possibly, with humans in order to performs its activities.

IAs, DOAs and AAs are autonomous, reactive and social, but they are not pro-active. In fact, all of them operate without the direct intervention of humans or other agents, react to the stimuli of the environment (i.e., the other agents and humans), interact with other agents and the IAs with humans too, but none of them show a goal oriented behaviour.

UPAs and SAs show all the properties of the weak notion of agent. In fact, a UPA starts pushing actions and a SA builds a team and modifies user queries with the goal to satisfy user interests. Moreover, a UPA shows additional properties that is the capability to learn user profiles: it builds user profiles from the information received by the IAs and uses such profiles to drive the IAs actions.

3.2 Agent architecture models

The weak notion of agent can be used to classify agent architecture models too. In fact, while some of these models are designed to offer complete reactive, pro-active and social agents, other models are oriented to offer task-oriented agents showing the balance among reactivity, pro-activity and social ability it needs to perform a particular kind of tasks.

We consider three main classes of agent architectures: reactive architectures, Belief-Desire-Intention architectures and layered architectures.

Reactive architectures do not have any internal, symbolic models of their environment, and they act using a

stimulus/response type of behaviour by responding to the present state of the environment in which they are embedded [4,10].

The behaviour of such architectures can be described through two functions. The first function, called *see*, maps input data from the environment to perceptions. The second function, called *action*, maps perceptions to actions. Figure 2 shows a schematic diagram describing the behaviour of such architecture.

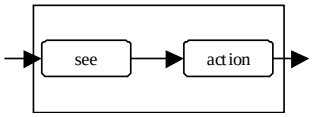

Figure 2. Schematic diagram of a reactive architecture.

The big advantage of reactive architectures is simplicity, however they have two main disadvantages: they cannot be pro-active because of their stimulus/response behaviour and they cannot learn from experience because of they have not an internal state.

Belief-Desire-Intention (BDI) architectures represent a system as a rational agent having certain mental attitudes of beliefs, desires, and intentions representing respectively the information, motivational and deliberative states of the system. These mental attitudes determine the system's behaviour and are critical for achieving adequate or optimal performance when deliberation is subject to resource bounds [3,18,22].

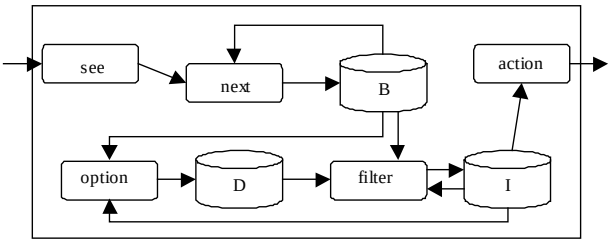

Figure 3. Schematic diagram of BDI architecture.

The behaviour of such architectures can be described through five functions. The first function, called *see*, maps input data from the environment to perceptions. The second function, called *next*, builds a set of new beliefs on the basis of current sets of perceptions and beliefs. The third function, called *options*, maps the new set of beliefs and the current set of intentions in a set of desires. The forth function, called *filter*, updates the set of intentions on the basis of the previous set of intentions and the current sets of beliefs and desires. The fifth function, called *action*, maps intentions to actions. Figure 3 shows a schematic diagram describing the behaviour of such an architecture.

BDI architectures are attractive for two main reasons. The first reason is that its behaviour is intuitive for humans. The second reason is that they present a clear functional decomposition indicating what sorts of subsystems might be required to build an agent. The main problem is to realise an efficient implementation.

Given the requirement that an agent should be capable of reactive and pro-active behaviour, an obvious decomposition involved the use of two separate subsystems to deal with these two different behaviours. This idea leads naturally to a class of architectures, called layered architectures in which the various subsystems are arranged into a hierarchy of interacting layers.

Layered architectures are divided in two subclasses: horizontal and vertical layered architectures. In horizontally layered architectures, the software layers are each directly connected to the sensory input and action output [4]. In vertically layered architectures sensory input and action output are each dealt with by at most one layer each. In particular, vertically layered architectures are divided into one pass control and two pass control architectures. In one pass control architectures, control flows sequentially through each layer until the final layer generates action output [11]. In two pass control architectures, information flows up the architecture and then control flows back down. Figure 4 shows the schematic diagrams of three main categories of layered architectures [20].

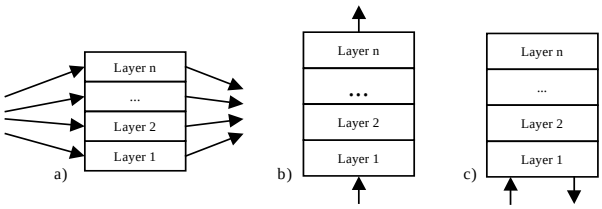

Figure 4. Schematic diagrams of three main categories of layered architectures: a) a horizontally layered architecture, b) an one pass control vertically layered architecture and c) a two pass control vertically layered architecture.

The great advantage of horizontally layered architectures is their conceptual simplicity to build an agent that exhibits some different types of behaviour. The problem is when these behaviours are in competition for some resources or to perform actions. In this case, it is necessary to introduce a mediator that implements a central control, but also introduces a bottleneck into the agent behaviour. Vertically layered architectures have not the problem of the central control and the way to pass control and information presents some similarities with the way that organisations work, but has a limited flexibility because control must pass between each different layer.

3.3 **Agent architectures**

At this point, we have the information to choose the most appropriate architecture for the different agent types we need in the multi-agent system for corporate memory management.

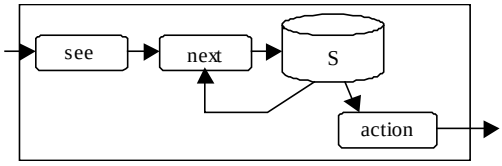

Figure 5. Schematic diagrams of a "reactive with state" layer of a horizontally layered architecture.

IAs, DOAs and AAs are not pro-active and the tasks they must perform can be easily defined eliminating possible conflicts on actions and resources. Therefore, a reactive and horizontally layered architecture, where each layer is dedicated to the execution of a particular task of the agent, seems to be the most

appropriate. However, it is not possible to use a purely reactive architecture because the execution of some tasks is influenced by history. Then, we can realise each layer extending the reactive architecture model to manage agent state: the behaviour of each layer, we call "reactive with state" layer, can be described through three functions. The first function, called *see*, maps input data from the environment to perceptions. The second function, called *next*, builds the new agent state on the basis of the current agent state and perceptions Finally, the third called *action*, maps action state to actions. Figure 5 shows a schematic diagram describing the behaviour of a "reactive with state" layer.

UPAs and SAs are pro-active; therefore a BDI or a vertically layered architecture seem to be the most appropriate architectures. From a first analysis, it seems that the tasks these agents must perform can be easily defined limiting the possible conflicts on actions and resources. Therefore, the BDI architecture seems to have some advantages because with few conflicts among agent tasks the work mainly consists in the mapping of agent tasks in agent intentions. However, the use of a vertically layered architecture is also possible with the limited effort to decompose the tasks on the basis of the competence of the different layers.

Some of the tasks that UPAs and SAs must perform are reactive. One of BDI architecture limits is that tasks performing agent intentions are executed by monolithic processes. Therefore, a BDI architecture does not optimally support reactivity, in a sense that it does not provide mechanisms allowing, for example, to recognise emergency situation in time and to stop current task to start the task managing this emergency, but delegates the implementation of such mechanisms within the individual tasks realising agent intention. This is not a big problem, because the reactive tasks performed by UPAs and SAs do not manage emergency situations and so they can wait for the end of the running task without any problem for the agent.

4 AGENT INTERACTION PROTOCOLS FOR CORPORATE MEMORY MANAGEMENT

The second step to realise the multi-agent system for corporate memory management is to identify the different types of protocols that allow the interaction between the agents of the system. The simplest protocols come from concurrency and distributed systems research; the others mainly come from distributed artificial intelligence research. From the task description of section 2, we identified three protocols of the first class: request, query and subscription protocols, and two of the second class: contract net and partial global planning protocols.

Request and query protocols are usually the simplest and the most used protocols in an agent system and respectively allow an agent to request the execution of an action or to request some information to another agent. In this case, the protocol is based on two agents exchanging two messages: an agent sends a request/query message and the other agent replies a response message.

Subscription protocol allows an agent to receive automatically some kinds of information from another agent when they become available to this agent. In this case, the protocol is based on two agents and a sequence of messages: an agent sends a subscription message, the other agent sends an *inform* message when it acquires new information related to the subscription and, finally, the first agent sends a message to remove its subscription.

In our multi-agent system the subscription protocol is used by UPAs that subscribe themselves to IAs and AAs respectively to receive information about user actions and to receive information about the new documents.

Contract net protocol allows an agent, called manager, to assign a task to one among a set of agents, called contractors, through a negotiation [23]. In this case, the following steps can describe the protocol:

- the manager announces a task to be performed to the potential contractors;
- the contractors respond either declining the offer or proposing a bid;
- the manager awards a contract to a contractor;
- the contract accepts the contract;
- the contract reports the results of the tasks.

In our multi-agent system the all the agents use the contract net protocol. In particular, IAs and SAs act as managers and UPAs, DOAs and AAs act as contractors.

Partial global planning protocol allows to build a global plan starting from the partial plans that a set of agents can independently generate [8]. In this case the protocol can be described by:

- an initial step, in which each agent sends a partial plan to the other agents, and
- an iterative step, in which each agent build a partial global plan and sends it to the other agents; the iteration ends when all the agents converge to the same partial global plan.

In our multi-agent system, if the corporate memory is distributed, the partial global planning protocol is used by SAs that build a global query from the partial queries generated by the different SAs.

5 AGENT ARCHITECTURES AND PROTOCOLS WITH JADE

The realisation of such a multi-agent system will be simplified through the use of an agent-oriented development environment.

5.2 JADE

JADE (Java Agent Development Environment) is a software framework to make easy the development of agent applications in compliance with the FIPA specifications [12] for interoperable intelligent multi-agent systems [1]. JADE is an Open Source project, and the complete system can be downloaded from JADE Home Page [15]. The goal of JADE is to simplify development while ensuring standard compliance through a comprehensive set of system services and agents. To achieve such a goal, JADE offers the following list of features to the agent programmer:

- FIPA-compliant Agent Platform, which includes the AMS (Agent Management System), the default DF (Directory Facilitator), and the ACC (Agent Communication Channel). All these three agents are automatically activated at the agent platform start-up.

- Distributed agent platform. The agent platform can be split on several hosts. Only one Java application, and therefore only one Java Virtual Machine, is executed on each host. Agents are implemented as one Java thread and Java events are used for effective and lightweight communication between agents on the same host. Parallel tasks can be still executed by one agent, and JADE schedules these tasks in a co-operative way.
- A number of FIPA-compliant additional DFs (Directory Facilitator) can be started at run time in order to build multi-domain environments, where a domain is a logical set of agents, whose services are advertised through a common facilitator.
- Java API to send/receive messages to/from other agents; ACL messages are represented as ordinary Java objects.
- FIPA97-compliant IIOP protocol to connect different agent platforms.
- Lightweight transport of ACL messages inside the same agent platform, as messages are transferred encoded as Java objects, rather than strings, in order to avoid marshalling and unmarshalling procedures.
- Library of FIPA interaction protocols ready to be used.
- Graphical user interface to manage several agents and agent platforms from the same agent. The activity of each platform can be monitored and logged. All life cycle operations on agents (creating a new agent, suspending or terminating an existing agent, etc.) can be performed through this administrative GUI.

The JADE system can be described from two different points of view. On the one hand, JADE is a runtime system for FIPA-compliant Multi Agent Systems, supporting application agents whenever they need to exploit some feature covered by the FIPA standard specification (message passing, agent life-cycle management, etc.). On the other hand, JADE is a Java framework for developing FIPA-compliant agent applications, making FIPA standard assets available to the programmer through object oriented abstractions. In this paper, we centre the discussion on the second aspect. A detailed discussion about the first aspect can be found in [1].

5.2 **JADE agent model**

JADE can be seen as a software framework for agent-based applications, with application programmers concentrating themselves on writing agents and on specifying each agent role within the agent society. Since complex tasks in multi agent systems are usually tackled using collaboration among many agents, a single agent is typically a strongly cohesive piece of software. On the other hand, asynchronous message passing with pull consumer messaging model grants a very loose coupling between different agents. Furthermore, no implementation inheritance (and no code reuse) is considered when dealing with software agents. In this way, software agents bear strong resemblance with actors, and indeed JADE execution model has its roots in actor languages.

The abstraction used to model agent tasks is the *Behaviour*: each JADE agent holds a collection of behaviours which are scheduled and executed to carry on agent duties. Behaviours represent logical threads of a software agent implementation. According to *Active Object* design pattern [19], every JADE agent runs in its own Java thread, thereby satisfying autonomy property; instead, in order to keep small the number of threads required to run an agent platform, all agent behaviours are executed co-operatively within a single Java thread. So, JADE uses a *thread-per-agent* execution model with co-operative intra-agent scheduling.

Using a single Java thread to handle multiple agent behaviours needs some sort of scheduling policy. JADE relies on a *"co-operative scheduling on top of the stack"*, in which all agent behaviours are run from a single stack frame without context saving (*on top of the stack*) and a behaviour continues to run until it returns from its main function and cannot be pre-empted by other behaviours (*co-operative scheduling*); of course ordinary pre-emption is still active between different agent threads and among JADE system threads: co-operative scheduling is strictly an intra-agent policy.

Using co-operative behaviours to model multiple agent conversation is a lightweight approach to concurrency, trying to achieve low latency by working entirely in user space. Similar techniques are customary in modern high performance network protocols and messaging libraries [7]. Besides, JADE model is an effort to provide fine-grained parallelism on coarser grained hardware. A likewise, stack based execution model is followed by Illinois Concert runtime system [17]; willing to provide a runtime environment for parallel object oriented languages, Concert can execute concurrent method calls optimistically on the stack, reverting to real thread spawning only when the method is about to block, saving the context for the current call only when forced to.

However, Concert system relies on compiler support to select concurrency strategy (really parallel or lazily stack based) and to save method context when needed. On the other hand, JADE is not a new language but a Java development framework: saving the context when an agent behaviour is blocking would put an heavy burden on JADE users, because they should write state saving code themselves, since no compiler would be going to write it for them.

Choosing not to save behaviour execution context means that agent behaviours start from the beginning every time they are scheduled for execution and local variables are reset every time. So, behaviour specific state that must be retained across multiple executions is to be stored into behaviour instance variables. A general rule for transforming an ordinary Java method into a JADE behaviour is the following:

1. *Turn the method body into an object whose class inherits from Behaviour.*
2. *Turn method local variables into behaviour instance variables.*
3. *Add the behaviour object to agent behaviour list during agent startup.*

The above guidelines apply the *reification technique* [16] to agent methods, according to *Command* design pattern [13]; an agent behaviour object reifies both a method and a separate thread executing it. While reification yields enhanced flexibility in task scheduling and execution, it often results in verbose program text (a problem commonly encountered in reflective code, too). A whole new class must be written and instantiated for each agent behaviour, and this can easily lead to programs hard to understand and maintain. JADE application programmers can compensate for this shortcoming using Java *Anonymous Inner Classes*; this language feature is similar to Smalltalk code blocks and makes the amount of code necessary for defining an

agent behaviour only slightly higher than for writing a single Java method.

Sometimes real intra-agent multithreading may seem unavoidable: for example, an agent acting as a wrapper onto a DBMS could issue multiple queries in parallel, or an agent might want to block on a stream or socket while still being able to engage in ordinary conversations. Really, this kind of problems occur only when an agent must interact with some non-agent software; FIPA acknowledges that these are boundary conditions for the execution model and deals with them in a separate part of the standard (namely, FIPA part 1 is about agent management and FIPA part 3 deals with external, non-agent software).

JADE *thread-per-agent* execution model can deal alone with all the most common situations involving only agents: this is because every JADE agent owns a single message queue from which all ACL messages are to be retrieved. Having multiple threads but a single mailbox would bring no benefit in message dispatching, since all the threads would still have to synchronise on the shared mailbox. On the other hand, when writing agent wrappers for non-agent software, there can be many interesting events from the environment beyond ACL message arrivals. Therefore, application developers are free to choose whatever concurrency model they feel is needed for their particular wrapper agent; ordinary Java threading is still possible from within an agent behaviour, as long as appropriate synchronisation is used.

5.3 Using behaviours to build complex agents

The developer who wants to implement an agent must extend the *Agent* class and implement the agent-specific tasks by writing one or more *Behaviour* subclasses, instantiate them and add the behaviour objects to the agent. User defined agents inherit from the *Agent* class the basic capability of registering and deregistering with their platform and a basic set of methods that can be called to implement the custom behaviour of the agent (e.g. send and receive ACL messages, use standard interaction protocols, register with several domains). Moreover, user agents inherits from their *Agent* superclass two methods: *addBehaviour(Behaviour)* and *removeBehaviour(Behaviour)*, which allow managing the behaviour list of the agent.

JADE contains ready made behaviours for the most common tasks in agent programming, such as sending and receiving messages and structuring complex tasks as aggregations of simpler ones. For example, JADE offers a so-called *JessBehaviour* that allows full integration with JESS [14]. JESS is a scripting environment for rule programming written in Java offering an engine using the Rete algorithm to process rules. Therefore, while JADE provides the shell of the agent and guarantees the FIPA compliance, JESS allows using rule-oriented programming to define agent behaviours and exploiting its engine to execute them.

Behaviour is an abstract class that provides the skeleton of the elementary task to be performed. It exposes three methods: the *action()* method, representing the "true" task to be accomplished by the specific behaviour classes; the *done()* method, used by the agent scheduler, that must return *true* when the behaviour has finished and can be removed from the queue, *false* when the behaviour has not yet finished and the *action()* method must be executed again; the *reset()* method, used to restart a behaviour from the beginning.

Because of the non pre-emptive multitasking model chosen for agent behaviours, agent programmers must avoid to use endless loops and even to perform long operations within *action()* methods. This is because when some behaviour's *action()* is running no other behaviour can proceed until the end of the method (of course this is true only with respect to behaviours of the same agent: behaviours of other agents run in different Java threads and can still proceed independently).

Moreover, since no stack context is saved, every time *action()* method is run from the beginning: there is no way to interrupt a behaviour in the middle of its *action()*, yield the CPU to other behaviours and then start the original behaviour back from where it left.

For example, suppose a particular operation *op()* is too long to be run in a single step and is therefore broken in three sub-operations, named *op1()*, *op2()* and *op3()*. To achieve desired functionality one must call *op1()* the first time the behaviour is run, *op2()* the second time and *op3()* the third time, after which the behaviour must be marked as terminated. The code will look like the following:

```
public class my3StepBehaviour {
  private int state = 1;
  private boolean finished = false;
  public void action() {
    switch (state) {
      case 1: { op1(); state++; break; }
      case 2: { op2(); state++; break; }
      case 3: { op3(); state = 1;
                finished = true; break;
              }
    }
  }
public boolean done() {
    return finished;
  }
}
```

Following this idiom, agent behaviours can be described as finite state machines, keeping their whole state in their instance variables.

When dealing with complex agent behaviours (as agent interaction protocols) using explicit state variables can be cumbersome; so JADE follows a compositional approach to allow application developers to build their own behaviours out of the simpler ones directly provided by the framework. Applying the *Composite* design pattern [13], *ComplexBehaviour* class is itself a *Behaviour,* but can have an arbitrary number of sub-behaviours or *children*; this class also defines the two methods *addSubBehaviour(Behaviour)* and *removeSubBehaviour(Behaviour)*, allowing to define recursive aggregations of several sub-behaviours. Since *ComplexBehaviour* extends *Behaviour*, the agent writer has the possibility to implement a structured tree composed of behaviours of different kinds (including *ComplexBehaviour*s themselves). The agent scheduler only consider the top-most tasks for its scheduling policy: during each "time slice" (which, in practice, corresponds to one execution of the *action()* method) assigned to an agent task only a single subtask is executed. Each time a top-most task returns, the agent scheduler assigns the control to the next task in the ready queue.

Besides *ComplexBehaviour*, JADE framework defines some other direct subclasses of *Behaviour*: *SimpleBehaviour* class can

be used by agent developer to implement atomic steps of the agent work. A behaviour implemented by a subclass of *SimpleBehaviour* is executed by JADE scheduler in a single time frame. Two more subclasses carry out specific actions: *SenderBehaviour* and *ReceiverBehaviour*. Notice that neither of these classes is abstract, so they can be directly instantiated passing appropriate parameters to their constructors. *SenderBehaviour* allows sending a message, while *ReceiverBehaviour* allows receiving a message which can be matched against a pattern; the behaviour blocks itself (without stopping all other agent activities) if no suitable messages are present in the queue.

The two classes *ComplexBehaviour* and *SimpleBehaviour* leave to their subclasses the choice of their termination condition and, for complex behaviours, the actual children scheduling policy. For atomic behaviours, two subclasses are provided: *OneShotBehaviour* is an abstract class that models behaviours that must be executed only once. *CyclicBehaviour* is an abstract class that models behaviours that never end and must be executed forever.

ComplexBehaviour class fulfils the responsibility of organising its children, but it defers children scheduling policy to subclasses; JADE provides ready made subclasses with common policies, but application programmers are free to implement their own.

SequentialBehaviour is a *ComplexBehaviour* that executes its sub-behaviours sequentially, it blocks when its current child is blocked and it terminates when all its sub-behaviours are done.

NonDeterministicBehaviour is a *ComplexBehaviour* that executes its children behaviours non-deterministically, it blocks when all its children are blocked and it terminates when a certain condition on its sub-behaviours is met. The following conditions have been implemented: ending when all its sub-behaviours are done, when anyone among its sub-behaviours terminates or when al least N sub-behaviours have finished.

JADE recursive aggregation of behaviour objects resembles the technique used for graphical user interfaces, where every interface widget can be a leaf of a tree whose intermediate nodes are special container widgets, with both rendering and children management features. An important distinction, however, exists: JADE behaviours are reifications of execution tasks, so task scheduling and suspension are to be considered, too.

Thinking in terms of software patterns [13], if *Composite* is the main structural pattern used for JADE behaviours, on the behavioural side we have *Chain of Responsibility*: agent scheduling directly affects only top-level nodes of the behaviour aggregation tree, but every composite behaviour is responsible for its children's scheduling within its time frame. Likewise, when a behaviour object is blocked or restarted a notification mechanism built around a bi-directional *Chain of Responsibility* scheme provides all necessary event propagation.

5.4 Interaction Protocols

Interaction protocols are used to design agents interaction providing a sequence of acceptable messages and a semantic for those messages. FIPA defines a set of general-purpose interaction protocols that provide a well-known interaction means to promote the social and cooperative nature of agents.

While each FIPA ACL message kind is given a formal semantics based on the speech-act theory, this is still not enough to satisfy the need for sociality of agent systems. This is because a typical agent interaction encompasses more than a single message, so more comprehensive abstractions are needed. FIPA specifications provide this abstractions as a collection of interaction protocols. When an agent engages in a conversation according to some interaction protocol, it can play two roles: the *initiator* role is the one who creates the conversation, by sending the first ACL message belonging to the new conversation; the *responder* role is the one who receives the first ACL message.

5.5 FIPA standard interaction protocols

The *fipa-request* protocol allows an agent to request another agent to perform some action; this is similar to ordinary request/response protocols used in client/server systems, but with a significant difference. Since software agents are autonomous entities, an agent can refuse to perform the requested action even if able to do so. So, while a client/server call either succeeds or fails raising an exception, a *fipa-request* interaction can succeed, can fail for a lack of the receiver-agent capabilities, but can also fail for the unwillingness of the receiver to perform the task at hand.

The *fipa-query* protocol is to be used to request an agent to provide some information to the protocol initiator. Since queries to an agent knowledge are not supposed to interact with external software systems, the *fipa-query* protocol shares the outcome set with the *fipa-request* protocol, but has a flat structure.

The *fipa-request-when* protocol is a variation of the *fipa-request* protocol: the initiator agent requests the responder agent to perform an action in the future, and after an initiator-supplied precondition is true. At that time the responder agent will try executing the action and will notify the initiator using an inform message if the attempt will have succeeded or using a failure message otherwise.

The *fipa-contract-net* protocol is the first of the higher level FIPA interaction protocols. An agent, called *manager*, can initiate a *fipa-contract-net* protocol sending a call for proposal, or *cfp*, message to a set of other agents; some agents will answer *not-understood* or *refuse* and will be discarded, but some others will hopefully send back a *propose* message, declaring their will to perform the requested action under a constraint set, such as price, time, or some other kind of constraint. The manager agent can then evaluate all the proposals and select one among them according to some criterion.

The *fipa-iterated-contract-net* protocol allows the manager to incrementally refine its call for proposal until a suitable contract is made. The only difference with respect to what previously described is that a manager can refuse all proposals, with *reject-proposal* messages, and issue a revised *cfp* using the same conversation identifier.

fipa-auction-english protocol resembles an auction for selling goods, led according to English style, using a low initial price and rising it gradually until no buyer declares its intention to pay. At that point, the last buyer can, and must, acquire the goods for sale paying the amount he or she accepted before.

The *fipa-auction-dutch* protocol realises the Dutch style for auctions, used for flowers market and working basically the opposite as the English auction: the auctioneer starts with a price much higher than the real market value of the goods he or she is trying to sell, then lowers the price gradually until one of the buyers accepts the suggested price and buys the goods.

5.6 Implementing interaction protocols with JADE

JADE implements the communication policies between agents by means of protocols defining the set of possible conversations between the agents, in particular, for every protocol that JADE supports a couple of classes is provided: one for agents playing the initiator role and another for responder agents. For each designed protocol, a semantic is also provided in order to let third-parties agents interact with the implemented ones. Besides providing FIPA interaction protocols, JADE allows the definition of new interaction protocols. However, the user is suggested to realise interaction protocols as a composition of FIPA protocols in order to ease their implementation with JADE and to support inter-operability with other FIPA-compliant systems.

To show how JADE supports FIPA interaction protocols, the *FipaRequestInitiatorBehaviour* class will be used as an example; this will also give a practical application of JADE concurrency model. This class must be used by agents willing to start new *fipa-request* conversations, asking some other agent to do something for them.

Looking at FIPA97 specification, one finds a graphical representation of the protocol, reported in Figure 6. The white boxes in the figure represent communicative acts that must be issued by the initiator agent, whereas the grey ones represent communicative acts issued by the receiver agent. So, from the initiator point of view, ACL messages corresponding to white boxes must be sent and the ones corresponding to grey boxes must be received. From the figure can also be seen that, after sending a *request* message, the initiator can expect either a *not-understood*, a *refuse* or an *agree*; in the last case, another message will arrive, i.e. either a *failure* (some problem occurred), or an *inform* (in two different flavours for actions without result and with one).

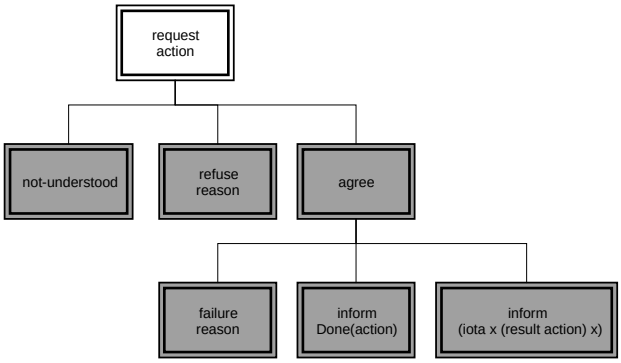

Figure 6. FIPA Request Interaction Protocol

Looking at the documentation of *FipaRequestInitiatorBehaviour* class, one finds a constructor accepting as parameter the ACL *request* message that will start the conversation; various abstract methods are provided to handle the various kinds of messages that can arrive during the protocol. Application programmers simply write their subclasses, implementing the abstract methods, named *handleAgree()*, *handleRefuse()* and so on. This approach relies on classic OO techniques such as abstract classes and tries to present a familiar programming style to application developers, but the inner workings of *FipaRequestInitiatorBehaviour* are a bit different.

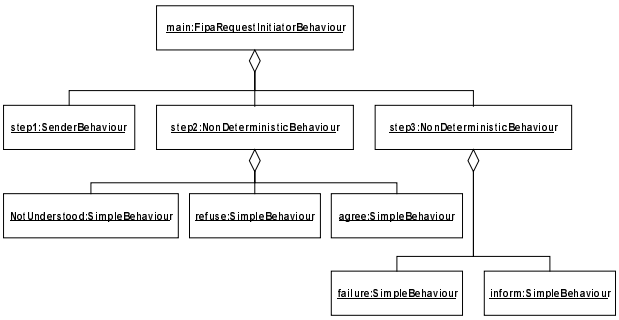

Figure 7. Internal structure of a JADE *FipaRequestInitiatorBehaviour* behaviour object.

This class relies on several private inner classes, which build an interaction protocol engine based on delegation, fully exploiting JADE intra-agent concurrency model. Looking again at Figure 5, the following algorithm can be written:

1. *Send the request message.*
2. *Receive the first reply and handle it.*
3. *If the first reply was an agree, then receive the second reply and handle it.*

When a complex behaviour is made by a list of steps, JADE library offers *SequentialBehaviour* class, which *FipaRequestInitiatorBehaviour* will be a subclass of. A *SenderBehaviour* can easily be used to carry out the first step, while the second one is more interesting.

The second step is a disjunction of three possibilities, seen as branches in the graph of Figure 6; it completes when any one among the three events (receiving a *not-understood*, receiving a *refuse* or receiving an *agree*) occurs. This observation allows to exploit the *NonDeterministicBehaviour* class to express the three possible branches as concurrent activities; the non deterministic behaviour object will be constructed so that it terminates as soon as one of its children receives something.

A similar approach is followed for the third step, but the second *NonDeterministicBehaviour* object will be added to the main *FipaRequestInitiatorBehaviour* by the handler of the *agree* message. All the atomic behaviours are implemented as *SimpleBehaviour* subclasses, which first receive their specific kind of message (using JADE pattern matching capabilities) and then invoke the handler method (e.g. *handleNotUnderstood()*). The overall object structure is drawn in Figure 7, showing also how similar the behaviour structure is to the original protocol structure in Figure 6.

5.7 Using JADE for building the multi-agent system for corporate memory management

From the previous description, JADE seems be an appropriate tool to realise our multi-agent system.

In fact, JADE agent model is based on a horizontally layered architecture, where each behaviour corresponds to a layer. Therefore, this agent model can directly realise the “reactive with state” and horizontally layered architecture, we need to

implement IAs, DOAs and AAs, if each behaviour implements the three functions *see*, *next*, and *action* described in section 3.3.

JADE does not offer a BDI architecture we need to implement UPAs and SAs; however, a BDI architecture can be implemented with a limited effort on the top of JADE agent model through an appropriate composition of a set of behaviours implementing the different functions defining the BDI architecture.

Finally, JADE offers an implementation of all the FIPA interaction protocols. Therefore, all the interaction protocols, except the partial global planning protocol, come with a limited effort from JADE. The realisation of the partial global planning requires a big effort, but can be guided by the implementation of FIPA interaction protocols.

ACKNOWLEDGEMENTS

We like to thank all the partners of the CoMMA project, because some of the ideas presented in this paper derive from discussions we have with them and from project documents they wrote for the project.

This research is partially supported by the European Commission through the contract *IST-1999-12217 - CoMMA - Corporate Memory Management through Agents.*

REFERENCES

[1] F. Bellifemine, A. Poggi and G. Rimassa. JADE - A FIPA-compliant Agent Framework. In Proc. Fourth International Conference on the Practical Application of Intelligent Agent and Multi Agent Technology (PAAM99), pp. 97-108, London, UK, 1999.

[2] B. Berney and E. Ferneley. CASMIR: Information retrieval Based on Collaborative User Profiling. 1999.

[3] M.E. Bratman. Intentions, Plans, and practical Reason. Haward University Press, Cambridge, MA, 1987.

[4] R.A. Brooks. A robust layered control system for a mobile robot. IEEE Journal of Robotics and Automation, 2(1):14-23, 1996.

[5] CoMMA. The CoMMA Project Home Page. Available at http://www.ii.atos-group.com/sophia/comma/HomePage.htm.

[6] O. Corby and R. Dieng. A CommonKADS Expertise Model Web Server. In Proc. Fifth Int. Symp. on the Management of Industrial and Corporate Knowledge, 1997.

[7] D. Dunning, G. Regnier, G. McAlpine, D. Cameron, B. Shubert, F. Berry, A.M. Merritt, E. Gronke and C. Dodd. The Virtual Interface Architecture. IEEE Micro, 18(2):58-64, 1998.

[8] E.H. Durfee. Coordination of Distributed problem Solvers. Kluwer Academic Press, Boston, MA, 1988.

[9] J. Euzenat. Corporate Memory through Cooperative Creation of Knowledge Base Systems and Hyper-Documents. In Proc. of Knowledge Acquisition Workshop (KAW'96), Banff, Canada, 1996.

[10] J. Ferber. Simulating with reactive agents. In E. Hillebrand and J. Stender (Eds.), Many Agent Simulation and Artificial Life, pp. 8-28, IOS Press, Amsterdam, Netherlands.

[11] I.A. Ferguson. Integrated control and coordinated behaviour: a case for agent models. In M. Wooldrige and N.R. Jennings, editors. Intelligent Agents: Theories, Architectures and Languages. LNAI-890, pp. 203-218, Springer Verlag, Berlin 1995.

[12] FIPA. FIPA Specifications 1997. 1997. Available at http://www.fipa.org.

[13] E. Gamma, R. Helm, R. Johnson and J. Vlissides. Design Patterns: Elements of Reusable Object Oriented Software. Addison Wesley, 1995.

[14] E.J. Fridman-Hill. Java Expert System Shell. 1998. Available at http://herzberg.ca.sandia.gov/jess.

[15] JADE. The JADE Project Home Page. 2000. Available at http://sharon.cselt.it/projects/jade.

[16] R.E. Johnson and J.M. Zweig. Delegation in C++. The Journal of Object Oriented Programming, 4(7):31-34, 1991.

[17] V. Karamcheti, J. Plevyak and A. Chien. Runtime Mechanisms for Efficient Dynamic Multithreading. Journal of Parallel and Distributed Computing, 37:21-40, 1996.

[18] D. Kinny and M.P. Georgeff. Commitment and effictiveness of situated agents. In Proc. 12th Int. Joint Conf. on Artificial Intelligence (IJCAI-91), pp. 82-88, Sidney, Australia, 1991.

[19] G. Lavender and D. Schmidt. Active Object: An object behavioural pattern for concurrent programming. In J.M. Vlissides, J.O. Coplien, and N.L. Kerth, Eds. Pattern Languages of Program Design. Addison-Wesley, Reading, MA, 1996.

[20] J.P. Muller, M. Pischel and M. Thiel. Modelling reactive behaviour in vertically layered agent architectures. In M. Wooldrige and N.R. Jennings, editors. Intelligent Agents: Theories, Architectures and Languages. LNAI-890, pp. 261-276, Springer Verlag, Berlin 1995.

[21] Rabarijaona, R. Dieng and O. Corby. Building a XML-based Corporate Memory. In Proc. European Knowledge Acquisition Workshop (EKAW'99). LNAI 1621, Springer-Verlag, 1999.

[22] A.S. Rao and M.P. Georgeff. Modeling rational agents within a BDI-architecture. In j. Allen, R. Fikes and E. Sandewall, editors. Proc. 2nd Int. Conf. on Principles of knowledge Representation and Reasoning, pp. 473-484, Morgan Kaufmann, San Mateo, CA, 1991.

[23] R.G. Smith. The Contract Net Protocol: High Level Communication and Control in a Distributed Problem Solver. IEEE Trans. on Computers, 29(12): 1104-1113, 1980.

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