Publicado en los Proceedings of The Practical Application of Intelligent Agents and Multi-Agent Technology, Londres, Apbril 1997.

Object-Oriented Agent Programming through the Brainstorm System

Analía Amandi[§] and Ana Price

Universidade Federal do Rio Grande do Sul - Instituto de Informática
Caixa Postal 15064 - Porto Alegre - RS - Brasil
[§]also: Universidad Nacional del Centro de la Provincia de Buenos Aires
Facultad de Ciencias Exactas - ISISTAN
San Martin 57 - (7000) Tandil - Buenos Aires - Argentina
email: {amandi,anaprice}@inf.ufrgs.br

Abstract

A multi-agent system can be viewed as an object-oriented system that has associated an intelligent meta-system. This point of view is based on the fact that an agent is basically an object that has behavioural capabilities, its own internal state (the mental state), and capabilities to interact with similar entities.

Two important characteristics must be added to objects to become agents: capabilities for taking their own decisions, and, capabilities for flexible manipulation of knowledge on treating mental states. This article is about an extension of object-oriented programming to support agent programming. The Brainstorm Meta-Architecture allows objects to become agents by associating to them a meta-level of agent capabilities, and a language combination of objets and logic permiting objects to express part of their knowledge through logic clauses. This extension has been developed on the Smalltalk-80 system.

Keywords: agent-oriented programming, meta-architectures.

1. Introduction

An object is an entity that has behavioural capabilities (implemented through methods) and internal knowledge (given through instance variables). An object-oriented system is composed by a set of object classes, whose instances (the objects) collaborate among themselves in order to achieve the system goals.

An agent is an autonomous entity that has behavioural capabilities (some of them, intelligent ones) and internal knowledge. A multi-agent system is composed by a set of agents (and possibly objects or other entities) that collaborate among themselves in order to achieve their local goals, which contribute together to achieve the system goals.

These definitions suggests the existence of common points between multi-agent systems and object-oriented systems, but the behavioural capabilities of their basic entities (objects and agents) are a lot different. While objects only define a well bounded behaviour, concerning to agents it can be distinguished five types de behavioural capabilities:

- Basic Behavioural Capabilities: basic actions that characterise a particular type of agents. For example, considering a robot as an agent, some of its basic behavioural capabilities can be: goToObject (anObject), goToDoor (aDoor), goTo (aPoint), pushObjectToObject (anObj1, anObj2), pushObjectToDoor (anObject, aDoor), etc.
- Communication Capabilities: those ones that determine the communication protocol among different agents. Communication capabilities depend on the selected communication performatives

(i.e. KQML performatives [Finin 93], COOL [Barbuceanu 95], etc.). For example, agents can have the following communication protocol: achieve (aClause), askIf (aClause), askAll (aQuery), etc.

- Perception Capabilities: these capabilities allow an agent to perceive certain facts without receiving any direct communication from the agents involved in such facts. Perception capabilities allow an agent to observe activities performed by other agents (a conversation among agents, transference of information between agents, destruction of agents, agents changing of communities, etc.) and environment changes (i.e. external sensors becoming enabled, locking of internal structures, etc.).
- Reaction capabilities: these allow agents to act without thinking in certain situations. Reaction capabilities allow an agent to react by performing an action when a given situation happens in a given context. The reaction can be fired because it has been detected an interesting situation. A situation is defined by received messages and/or perceived events.
- Deliberation Capabilities: such as, deciding which is the next action to be executed, or whether the agent will collaborate with another agent in a given task, etc.

Uniform communication among different types of agents, perception of interesting situations without passing messages among the involved parts, reaction to given situations, reasoning capabilities that permit to decide what to do next are not inherent capabilities to objects. The Brainstorm Meta-Architecture has been defined to permit objects to become agents by associating to them a meta-level of agent capabilities.

Also, agents have to represent in their mental state, their beliefs, goals, intentions, knowledge, mechanisms to select the next action, etc. To this purpose, it is convenient to have a language that permits both to express unambiguously knowledge and to infer sentences. Thus, the Logic Paradigm emerges as a good alternative.

It has been considered to manage internal knowledge expressed by logic formulae in the meta-architecture. Thus, an object that becomes an agent can express its knowledge by objects and logic clauses. For this goal, we have defined also an extension of Prolog to manage beliefs, goals and intentions.

In this article, we claim that a multi-agent system can be considered as an object-oriented system that has associated to it a meta-system allowing agent behaviour. Section 2 presents how objects become agents through the Brainstorm Meta-Architecture. Sections 3 explain each component of Brainstorm. Finally, it describes related works and presents the conclusions of this work.

2. How objects become agents

A multi-agent system can be considered as an object-oriented system that has associated to it an intelligent meta-system. By this way, an agent is viewed as an object that has a layer of intelligence, comprising a number of capabilities such as uniform communication protocol, perception, reaction and deliberation, all of them not inherent to objects.

The Brainstorm meta-architecture has been developed to extend objects for supporting agent behaviour. In this environment, a multi-agent system is defined as a reflective system [Maes 87], that is causally connected to the object-oriented system defined in the base level. The causal connection is established when an object receives a message: the message is intercepted by the meta-level, which decides what to do with it. For example, an agent can decide to deny a request of another agent.

The meta-architecture has been built using a meta-object approach [Maes 87]. An agent is defined by an object and a set of meta-objects. Each meta-object incorporates agent capabilities to simple objects. The capabilities that can be added to objects are:

- Uniform communication through a communication language. A *communication meta-object* defines a communication protocol among agents.
- Perception of events taking place in the environment. A *perception meta-object* observes the behaviour of an agent or an object, recording any event of interest. For example, agent A would like to observe the communications received by agent B. Then, agent A defines a perception meta-object for B, indicating what it shall observe.
- Reaction to given situations. When an agent either perceives something (through one of its perception meta-objects) or receives a message (intercepted through its communication meta-object), situations of interest can be detected. In this case, the agent can react executing a given action. A *reactor meta-object* is responsible for this behaviour.

- Deliberation. The deliberation process of an agent involves carefully thinking before making a decision about what to do next. A *deliberator meta-object* is responsible for this behaviour.
- Treating of knowledge using logic formulae. If an object needs to manage logic formulae, it shall have a *knowledge meta-object* associated to itself.

By selecting a set of these meta-objects to an object class, it is possible to define different agent types: reactive, deliberative and hybrid agents. Also, in the present Smalltalk-80 implementation, each meta-object can be specialised, allowing different behaviour for each agent type. Figure 1 shows the Brainstorm meta-architecture. We can observe that an object has several meta-objects in different levels. In the first level, three types of meta-objects of the base object can be defined: perception, communication and knowledge meta-objects. In the second level, two types of meta-objects of the situation manager can be defined: reactor and deliberator meta-objects.

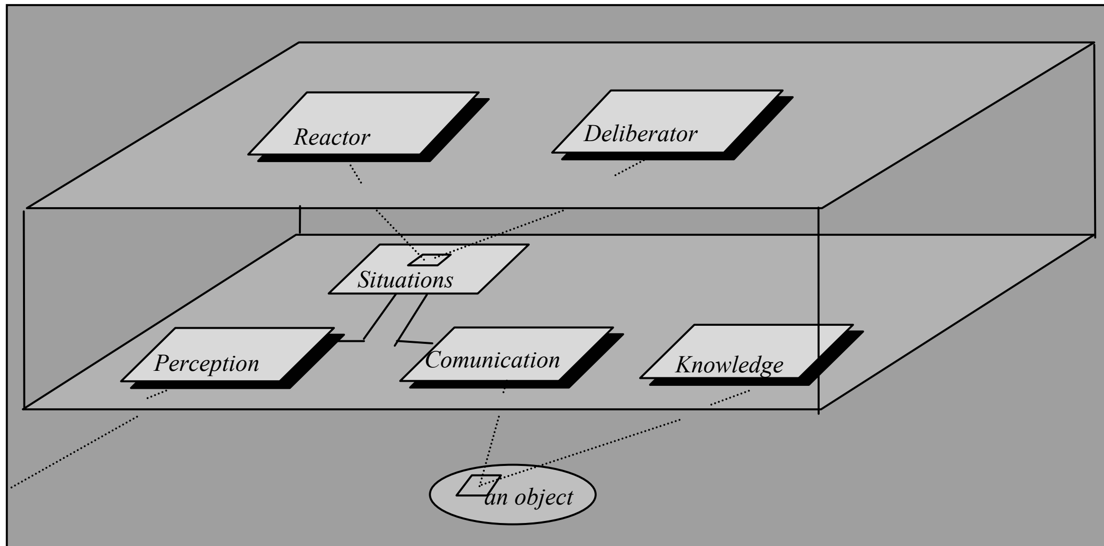

Figure 1 - The Brainstorm Meta-Architecture.

The first meta-level of the architecture works on messages received by the base level, and the second meta-level works on communication performatives processed by the communication meta-object and on interesting situations discovered by the situation manager. Each meta-object of a layer interferes in tasks executed by meta-object of the lower layer, changing thus the results of these tasks.

The transformation of a typical object in an agent is done by defining one knowledge meta-object, one communication meta-object, a set of perception meta-objects (possibly empty), and one reactor and/or a deliberator meta-object.

A multi-agent system is composed by traditional objects and agents. The communication system (messages) is uniform, permitting objects and agents work together. An important point is that agents define two types of basic capabilities for communicating and for acting. The basic communication capabilities permit the interaction between agents and objects. The action capabilities define the basic behavioural elements which agents can act in their environment.

When a developer wants to transform an object class in an agent class, he has first to indicate this situation and then to define the kind of knowledge the class will manage, whether it will represent a reactive, deliberative or hybrid agent, the communication language, situations of interest that are common to all instances, whether the instances will use an access password, common reactions, preconditions and postconditions of each method, etc. Figure 2 shows a template to enter part of this information. The example refers to a bank application. Through the book, it is possible to enter general data to all Bank instances: situations, knowledge types, selected communication language, what objects or agent to observe, when and how to react, method preconditions and postconditions, goals, intentions, beliefs, and how to act with other agents.

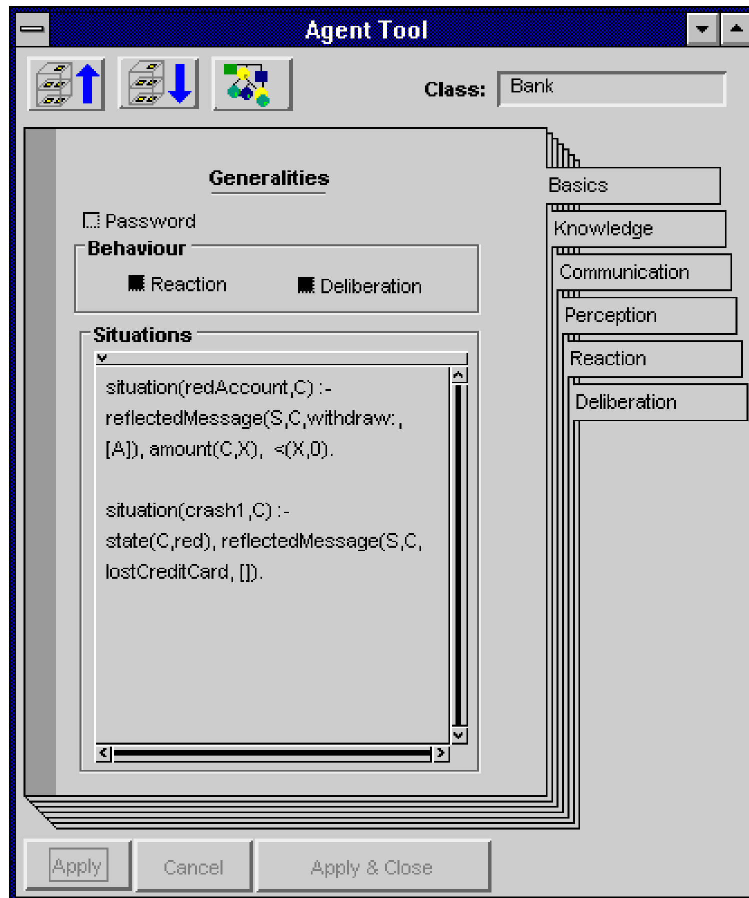

Figure 2 - General characteristics of an agent class.

When an object class is defined as an agent class, a meta-object (an instance of the MetaAgent class) is associated to the object class. Thus, each instance creation is intercepted and so the meta-level is built. Figure 3 shows the interception of the method *new* by a meta-object that is responsible for creating the meta-level of the new object. This meta-object creates the meta-objects needed for satisfying the agent specification. Thus, it is possible to define different types of agents in the same system.

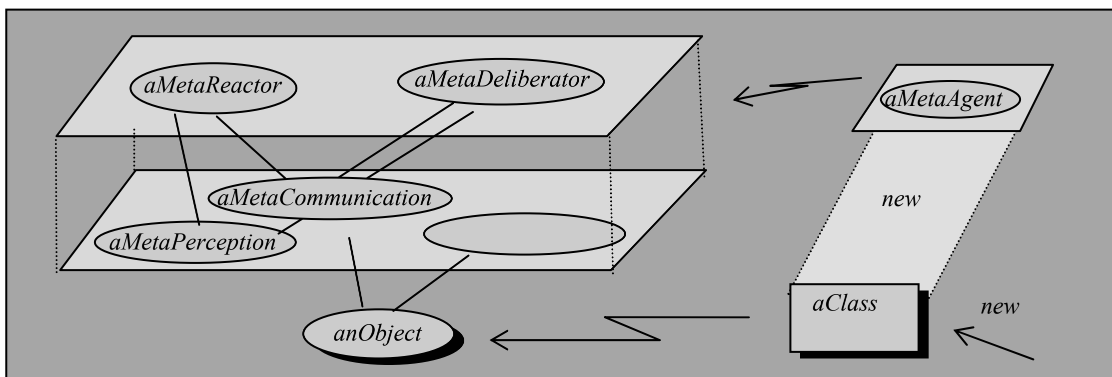

Figure 3 - An object class becomes an agent class.

Agent instances usually define their own interesting situations, reactions, reasoning forms, etc. These particularities are supported by their meta-objects. Figure 4 shows how to define this information.

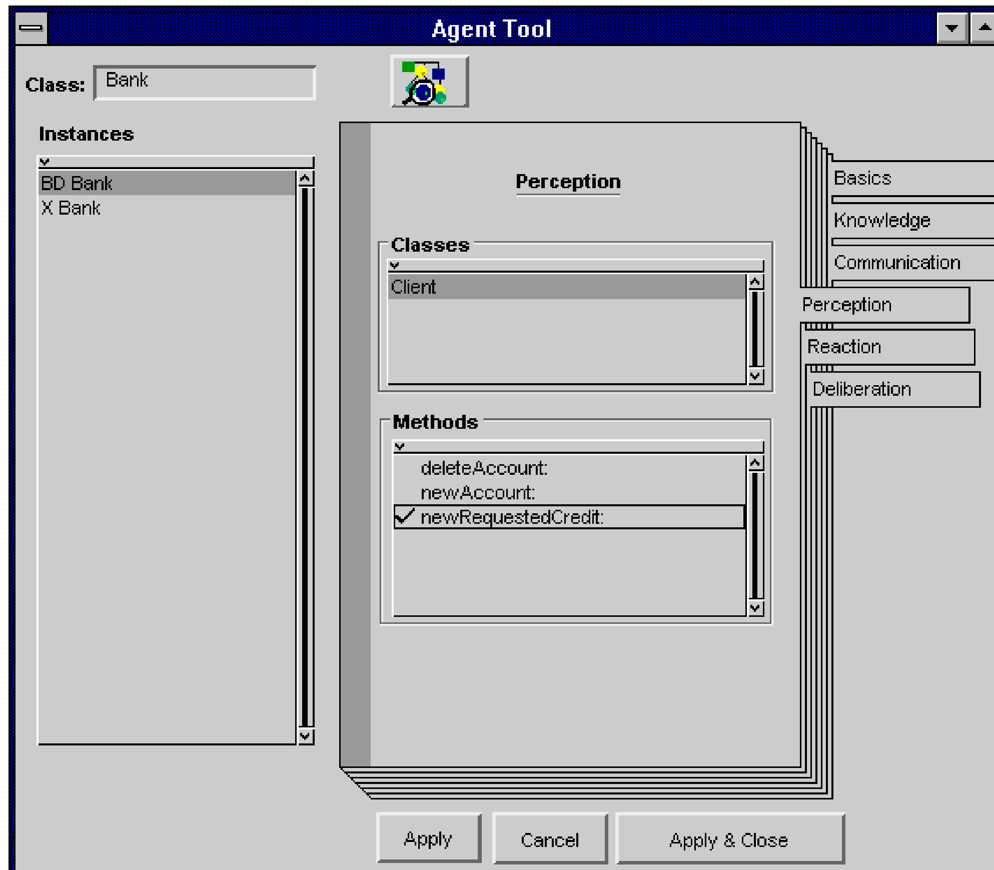

Figure 4 - Particular characteristics of an agent.

Agents shall define general characteristics of each instance in the meta-object (a MetaAgent instance) of its class. Each agent can record its own characteristics in its own meta-objects. For example, if banks are defined as agents, the Bank Class has a MetaAgent instance associated to it, which records interesting situations, reactions, reasoning details for all banks. For particular characteristics of each bank, each situation manager records its interesting situations and particular perceptions, each meta-reactor records its particular reactions, and each meta-deliberator records its reasoning particularities. The characteristics recorded for all instances of a class are inherited by its subclasses.

The next subsections present in detail each component of the meta-architecture, and how object and logic are combined for supporting agent complex mental states.

3. The Meta-Objects of the Brainstorm Architecture

3.1. The communication meta-object

A communication meta-object defines a communication language that may be adopted by agents. Basic actions shall be defined in the object class and the uniform communication protocol must be defined in a communication meta-object class. For example, for an agent Bank we can define basic actions such as 'evaluate a credit', 'open a bank account' in the Bank class. To allow a uniform protocol among agents, the bank must associate an instance of the KQML class (a MetaCommunication subclass) to him.

When an agent receives a message (see Figure 5), this message is reflected by the agent communication meta-object through a meta-object mechanism which actives the method "*handleMessage*:" with the reflected message as argument. This message is then analysed by the communication meta-object. If the message has been defined either in the object class as an acting capability or in its communication meta-object class (by the communication language), then it is registered for a future manipulation. In this occasion, the deliberator meta-object reflects the message

"*record*:" to define the message priority. For doing so, the deliberator uses its own rules. For instance, if the message sender is the boss, it has the most high priority of attending.

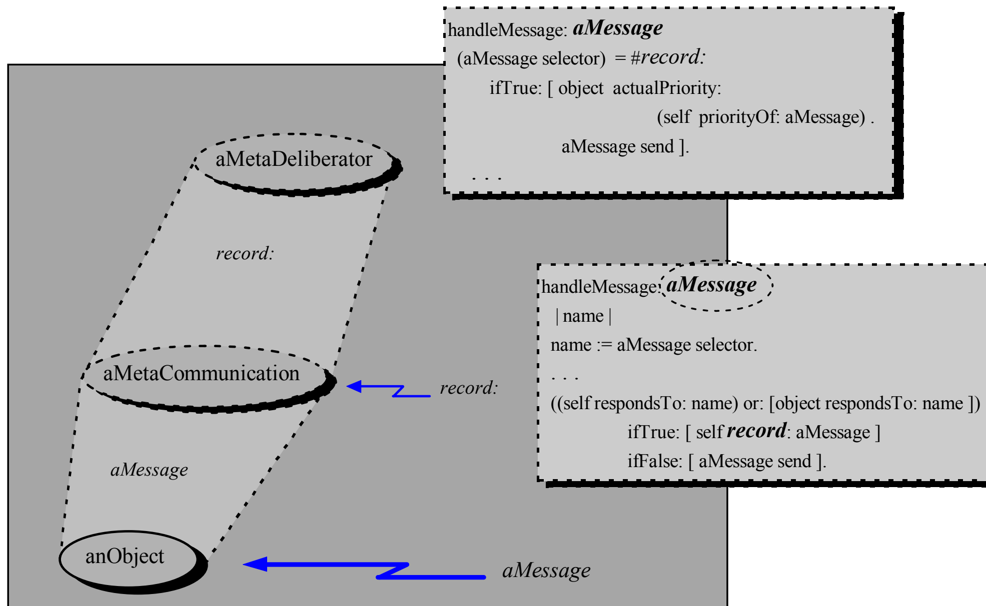

Figure 5 - An agent receives a message.

For example, if an agent receives a message "*achieve: aGoal*", this message is intercepted by the agent communication meta-object. This meta-object recognises the message as part of its communication language and invokes a method to record it for future attending. The deliberator meta-object intercepts the message "*record: aMessage*" to define a priority to it. Suppose the agent decides to give a media priority because it is in debt with the message sender and the fact of achieving that goal is not disadvantageous for him. Then, the deliberator meta-object permits to continue of the execution of the method "*record*:" of the communication meta-object, which records the message plus its corresponding priority.

The received messages can also collaborate to identify a situation of interest. This case is treated in the next section.

3.2. The perception meta-object

An agent can observe events taking place in its environment. The environment in the present proposal is composed by agents and objects, and so the message-passing is the only source of events. These events can draw a situation of interest for an agent.

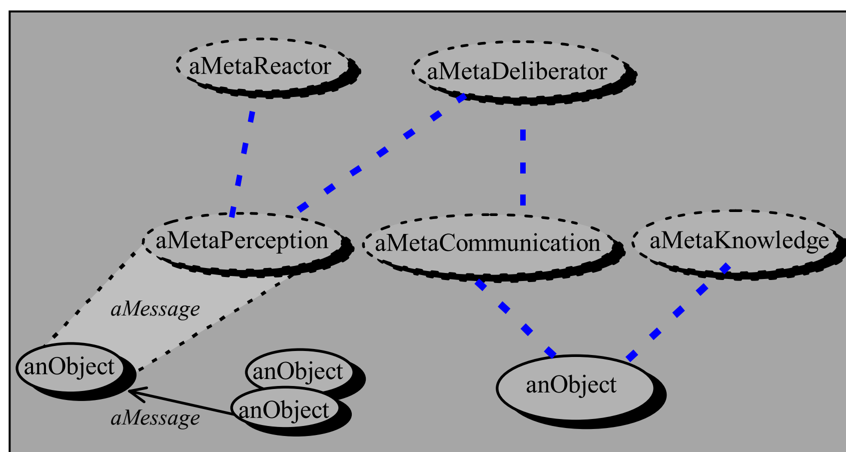

Figure 6 - An agent perceives a situation of interest.

To perceive events, agents shall include definitions of percepting meta-objects (instances of MetaPerception class). These meta-objects observe other agents or objects, reflecting their received messages (see Figure 6). This information (plus information about the messages received by the own agent) is evaluated by the situation manager searching for interesting situations. Whenever this type of situation is detected, the method "*newSituation: aStuation*" is invoked, being reflected by the meta-objects MetaReactor and MetaDeliberator. These last meta-objects decide how to treat the situation.

For example, an agent Bank, that has a lot of credit requests to evaluate, may want to know when the clients involved with those credits make big transactions of money. The bank wants to be more careful on the evaluation of those credit requests from clients that have made suspicious transactions. The object bank could ask to each object client for information (through a message), but a better solution would be to put a sensor in the accounts of those clients to observe their transactions. This design decision is based on the fact that this behaviour is a particular decision and a temporal policy of the bank. The implementation of this sensor is very simple: the agent bank associates a perception meta-objet to the accounts of clients (which can be either objects or agents) requesting a credit to observe the reception of the message "*withdraw: amount*". Whenever one of these accounts receives this message, the perception meta-object takes the message and sends it to its situation manager. The manager analyses whether *withdraw amount* is a suspicious transaction. If it is the case, it sends a message "*newSituation*:" to itself, which is reflected by the bank's deliberation meta-object that evaluates the requested credits of that client with more care.

3.3. The reactor meta-object

The reactor meta-object has the functionality of reaction to given situations. A situation is defined by received messages and perceived events. For example, an agent Bank has a set of credit requests. This bank has the policy of rejecting all credit request of clients that are delayed in the payment of other credits. Whenever states of old credits are analysed and a significant delay is detected, a situation of credit rejection is detected. For such a purpose, it is necessary to define both the case as an interesting situation and the reaction to this situation.

The situation is defined for this particular bank instance. It is called *creditRejection* and is associated to the object Client that delay a payment of a credit (the variable *Client* in the definition bellow). The situation is detected when: the perceived message is *delayedCreditPayment: for:* and the argument of this message (the time of delaying) is greater than three months.

The definition of this situation is:

situation(creditRejection, Client) :-
reflectedMessage(Sender, Receiver, delayedCreditPayment:for:, [Client,Time]),
>(Time,3).

Situations of interest shall be defined to be treated by reaction and/or deliberation. In the reactor, it is defined which situation determines which reaction. The reaction to the situation *creditRejection* is to reject credit requests of the client C. It means to invoke a method of the Client class denominated "*rejectAllCreditRequest*", which has been defined as a basic action in that class.

3.4. The Deliberator meta-object

The deliberator meta-object determines the acting form of agents. Different acting forms are defined in different subclasses of the MetaDeliberator class. An agent acting form is determined by the manner in which it has to act when: (a) it perceives a situation of interest, (b) it receives a message, and (c) it decides to do something.

When an agent perceives a situation, it can either analyse the situation immediately or give a priority for later analysis. When an agent receives a message, it can either attend the message immediately or give a priority for future attending it. For this reason, it is necessary to define when a situation or a message is treated immediately and the attending priority in case of leaving it for later. These facts can be defined in the class whether it is the same for each agent of the class, or defined in each agent whether the behaviour is unique to each one. This form of agent acting is concretised by:

attendSituation(aSituation) :- .... *attendMessage(aMessage) :- ....*
situationPriority(aSituation,P) :- ...... *messagePriority(aMessage,P) :- .....*

Attending a message and analysing a situation are activities that can be very different for each agent. For this reason, there are two different methods in the MetaDeliberator class for managing the perceived situations and the received messages; both methods can be redefined. Some details such as when the agent shall answer, deny or negotiate the answer of a message, are recorded in rules (either in the MetaDeliberator subclass or in each instance, depending on the necessities).

An agent also act without the necessity of any communication or perception of situations. A quantum of time is given to each agent for acting freely. Agents use this time for treating pending communications, making plans to achieve their goals (using the available planning algorithms), analysing their old plans, goals and intentions, and executing actions in their environment. Therefore, it is necessary to have defined priority rules for guiding the decisions about what to do next.

3.5. The knowledge meta-object

This component is responsible for the mental states of agents. It incorporates facilities for managing knowledge expressed as clauses. This capability is possible since an integration of an object-oriented language (Smalltalk-80 in this case) with the language Prolog has been implemented. Section 8 presents this integration.

The kind of clauses this component can manage is selected among a set of possibilities. Up to now, it is possible to use either classical logic programming (like Prolog), or temporal logic programming [Abadi 89] [Orgun 92], or an extension of Prolog for expressing beliefs, goals and intentions. Extensions to the Prolog interpreter can be easily incorporated, since the knowledge meta-object administrates the integration.

The knowledge meta-object permits the combination of different logic modules referred by instance variables and methods and make it temporally available for use.

The objective of combining objects and logic is to provide a support for allowing objects to be able to define and refer to logic knowledge representation in methods and instance variables. This fact does not imply that objects cannot represent part of their own knowledge through objects. For this reason, objects can be used in clauses as basic components.

The combination of objects and logic has been done using Smalltalk-80 and an implementation of Prolog [Aoki 94] for Smalltalk-80. Two points have to be taken into account here: a method may include logic clauses and a logic clause may include objects as arguments. When a method includes clauses, these must be enclosed by the symbols "{{" and "}}" and the variables used in queries must be declared. When a clause refers to objects as constants, the symbols "{" and "}" are used for enclosing the variables (instance, temporal, or class variables) that refer to the objects.

The combination of objects and logic clauses allows objects to:

- define logic modules as instance variables.
- define logic modules in methods.
- refers to logic modules in methods.
- represent knowledge by logic modules and objects.
- inherit logic modules defined in methods.

It is possible to record logic theories as instance variables. This means that objects can define distinct logic modules (in different instance variables), which can be used (or made available) in different contexts. In order to make available a logic module referred by a variable, the object has to send a message to itself indicating this situation.

For instance, an object of the Bank class can define different rules for credit evaluation in different logic modules (recorded in different instance variables). Depending on the case, the object Bank can make available the evaluation rules that consider appropriate (invoking the associated variable to that module). Thus, methods of this class can consult the available knowledge base, for example, for evaluating a given credit of a given client.

Also, logic modules may be defined in methods and may be referred by all objects of the class. For instance, if all clients are submitted to common rules for credit request evaluation it is convenient to define the rules in a method. The following method defines the common rules that banks may use for evaluating credit requests of its clients.

```
creditEvaluation
{{ creditEvaluation({aCredit}, {aClient}, 'Accept') :-
       history({aClient}, acceptable), send({aClient}, segurityFor:, [{aCredit amount}],S), >(S,90).
   creditEvaluation({aCredit}, {aClient}, 'Accept') :-
       realGuarantee({aClient},G), >(G,{aCredit amount}).
   creditEvaluation({aCredit}, {aClient}, 'Reject').
}}
```

To use internal knowledge defined by either variables (variables referring to logical knowledge) or methods, a developer must first make the knowledge available (using the associated variable or invoking the method) and then it is able to make queries. For example, a method for evaluating a credit for a client can be implemented by the following code:

```
evaluationOfCredit: aCredit  for: aClient
   | r |
   self creditEvaluation.
   {{ ?-creditEvaluation({aCredit}, {aClient}, R).  }}
   self print: ((aClient name), (aCredit number), r)
```

This method receives two objects as arguments: a credit (instance of Credit class) and a client (instance of Client class). It makes the clauses availabels in the method *creditEvaluation* (through *self creditEvaluation*), consults the knowledge base using the objects credit and client as constants and then prints the result (client's name, credit's number and evaluation result).

As it has been shown previously in the examples, clauses may use objects as constants and may send messages to objects from their own body. Thus, complex structures and entities with their own behaviour can be represented as objects and then may be used in clauses as constants.

Regarding to logic modules inheritance, it is possible to inherit knowledge defined in methods. As the methods of a class may include several clauses, a subclass can redefine such methods, excluding the inherited clauses or adding new clauses (by invoking first the redefined method). This is possible since a method is not a clause, and thus, the inclusion and exclusion of clauses are manipulated in separated ways.

The capability that objects can define and refer to logic modules has been implemented through meta-objects. If an object needs to manage logic knowledge, it has to be associated to a knowledge meta-object as explained above.

By the other hand, expressing beliefs, goals, and intentions are important for guiding actions. Plans generation is motivated by the satisfaction of intentions. Any changing of intentions shall modify plans. Any changing of beliefs and goals shall modify intentions. Intentions help to decide what to do next. For this reason, the implementation of Prolog for Smalltalk-80 [Aoki 94] has been first extended with temporal operators (NEXT, ALWAYS and EVENTUALLY) and then extended for expressing beliefs, goals and intentions.

For such purpose, we have taken as a base the formal theory of rational action of Cohen and Levesque [Cohen 90]. Using this theory, it is possible express beliefs, goals and intentions using the following syntax: BEL(temporalClause), GOAL(temporalClause), INTENTION(temporalClause).

For example, *GOAL(next 5 evaluated(aCreditRequest))*, defined in an instance of the Bank class, indicates that the bank has as a goal to have evaluated the requested credit *aCreditRequest* after five units of time. If this goal becomes an intention, a plan is generated (using the planning algorithms available in the deliberator meta-object) for satisfying the clause *next 5 evaluated(aCreditRequest).*

For selecting actions, shall be used the operators DONE(action) and HAPPENS(action) which indicate the action to be done and the action that happens next, respectively.

This extension of the Prolog interpreter checks beliefs inconsistency when new beliefs are added and whether a goal or an intention has to be ruled out.

4. Related Work

Several proposals for supporting the development of agents can be found in the literature. Languages and architectures have been proposed: languages such as Agent0 [Shoham 91], DAISY

[Poggi 95], AgentSpeak [Weerasooriya 95], MetateM [Fisher 94], CooL [Kolb 95], etc.; and agent architectures such as TouringMachines [Ferguson 92], InteRRaP [Müller 94], etc.

Some of the proposed languages had been based on object-oriented features, building agent types directly as classes. Daisy [Poggi 95] defines a class for each agent type. Each agent has a knowledge base, a set of actions and an engine for managing its behaviour. The knowledge base is represented by a set of relations. AgentSpeak [Weerasooriya 95] also defines a class for each agent type. Each agent has a database and services. The database contains a set of relations. The language CooL [Kolb 95] defines a class for each agent type and the knowledge is defined in terms of associations. The language MetateM [Fisher 94] uses object-oriented features defining an object as a set of rules.

Building an agent type directly as a class implies that the reasoning process has to be part of the class itself. The common characteristics and behaviour of agents could be defined either in a superclass Agent or in each new agent class. The first solution is rigid since only one class (the Agent class) cannot abstract intelligent behaviour that the diversity of agent types could need. In the second solution, all agent aspects must be implemented in each agent class, resulting so much effort for developers.

On the other hand, agent architectures provide good designs for building agents. But, few of them show flexibility (i.e.[Hayes-Roth 95]) and manipulation of complex mental states (i.e.[Fischer 96]).

In contrast, the Brainstorm architecture combined with an integration of objects and logic provides a clear separation between reasoning capabilities and basic behaviour, and provides a logic representation of mental states permitting to infer the next actions of agents in a natural manner. This approach permits the building of flexible and adaptable multi-agents systems on existent object-oriented systems.

5. Conclusions

We have presented an extension of object-oriented programming for developing multi-agent systems. This extension is based on the Brainstorm architecture, which proposes the definition of agent capabilities in a meta-level of objects.

These ideas has been implemented in Smalltalk-80 using the meta-object framework MOMF [Campo 96] and a Prolog interpreter developed for Smalltalk-80 [Aoki 94]. The implementation allows easy specialisation of meta-objects for building particular agent types.

6. References

[Abadi 89] Abadi M., Manna Z. Temporal Logic Programming. Journal of Symbolic Computation. vol. 8, pp.277-295. 1989.

[Aoki 94] Aoki A., et al. Mei-Prolog. Available by ftp: ftp.cs.man.ac.uk.

[Barbuceanu 95] Barbuceanu M., Fox M. COOL: A Language for Describing Coordination in Multi Agent Systems. Procedings of the First International Conference on Multi-Agent Systems, ICMAS'95. pp.17-24. June, 1995.

[Campo 96] Campo M., Price T. Meta-Object Manager: A Framework for Customizable Meta-Object Support for Smalltalk-80. Proceedings of the Brazilian Symposium of Languages. Brazil, september, 1996.

[Cohen 90] Cohen P., Levesque H. Intention is Choice with Commitment. Artificial Intelligence, vol.42, no.2/3, march 1990.

[Ferguson 92] Ferguson I. TouringMachines: Autonomuous Agents with Attitudes. Computer, p.51-55, may, 1992.

[Fisher 94] Fisher M. A Survey of Concurrent METATEM - The Language and its Applications. In D. M. Gabbay, H.J. Ohlbach, editors, Temporal Logic - Proceedings of the First International Conference. LNAI volume 827, Springer-Verlag, July 1994.

[Fischer 96] Fischer K., Müller J., Pischel M. A Pragmatic BDI Architecture. In Wooldridge M., Müller J. and Tambe M., editors, Intelligent Agents II. Lecture Notes in Artificial Intelligence no.1037, January, 1996.

[Finin 93] Finin T., et. al. Specification of KQML Agent Communication Language, The DARPA knowledge sharing initiative, External Interfaces Working Group.

[Hayes-Roth 95] Hayes-Roth B. An architecture for adaptive intelligent systems. Artificial Intelligence, vol.72, no.1-2, pp.329-365, January, 1995.
[Kolb 95] Kolb M. A Cooperation Language. Procedings of the First International Conference on Multi-Agent Systems, ICMAS'95. pp.233-238. June, 1995.
[Maes 87] Maes P. Concepts and Experiments in Computational Reflection. Proceeding of OOPSLA'87. 1987.
[Müller 94] Müller J., Pischel M. Modeling Interacting Agents in Dynamic Environments.
Proceedings of the European Conference on Artificial Intelligence, ECAI'94. John Wiley & Sons, p.709-713, 1994.
[Orgun 92] Orgun M., Wadge W. Theory and practice of temporal logic programming. Intensional Logics for Programming. Oxford Science Publications. 1992.
[Poggi 95] Poggi A. DAISY: An Object-Oriented System for Distributed Artificial Intelligence. In Wooldridge M. and Jennings N., editors, Intelligent Agents. Lecture Notes in Artificial Intelligence no. 890, January, 1995.
[Shoham 91] Shoham Y. AGENT0: A simple agent language and its interpreter. Proceedings of the AAAI'91, July, 1991.
[Shoham 93] Shoham Y. Agent-Oriented Programming. Artificial Inteligence, vol.60, no., pp.51-92, 1993.
[Weerasooriya 95] Weerasooriya D., Rao A. Ramamohararao K. Design of a Concurrent Agent-Oriented Language. In Wooldridge M. and Jennings N., editors, Intelligent Agents. Lecture Notes in Artificial Intelligence no. 890, January, 1995.