

DIMA

Zahia Guessoum

Team Objects and Agents for Simulation and Information Systems (OASIS)

Laboratoire d'Informatique de Paris 6 (LIP6)

8 rue du Capitane Scott, 75015 PARIS

e-mail: dima@poleia.lip6.fr

<http://www-poleia.lip6.fr/~guessoum/DIMA.html>

Contents

1	Introduction	3
2	Frameworks	3
2.1	ATN-Based Framework	4
2.2	Rule-Based Framework	6
3	A Generic Agent Architecture	7
3.1	A Modular Agent Architecture	8
3.2	Agent Behavior	9
3.3	Agent Meta-Behavior	9
3.4	Agent Engine	11
3.5	Communication and Interaction	12
4	Distribution and Replication	13
5	Overview of DIMA	14
6	Main Step to Develop a Multi-agent System	14
7	Conclusion	15
8	License	16
9	Third-party software	16

1 Introduction

DIMA is a Java multi-agent platform built as an extension of Object-Oriented Programming (OOP). It provides a generic and modular agent architecture with several facilities (lifecycle management, message passing, distribution, ...), and allows high heterogeneity in agent architectures (reactive, deliberative and hybrid), and various customizations.

DIMA is an attempt to provide answers to the basic questions on how to build a bridge between: 1) the implementation and modeling requirements of multi-agent systems [1] and 2) the implementation and modeling facilities and techniques provided by OOP [2]. These basic questions are:

- what is a generic structure to define the main features of an autonomous agent (pro-activity, sociability, ...)?
- how to accommodate the highly-structured OOP model into this generic structure [2]?

DIMA provides therefore:

- a modular agent architecture,
- a set of frameworks that implement well known and useful paradigms;

This report describes the main properties and components of DIMA. It also describes the design and development approach of DIMA. It is organized as follows: section 2 two presents the main provided frameworks. Section 3 describes the agent architecture. section 4 presents the distribution mechanism. Section 5 gives an overview of DIMA and Section 6 summarizes the development steps.

2 Frameworks

To facilitate the reuse of existing paradigms, DIMA provides several Frameworks:

- ATN-based framework (ATN for Augmented Transition Network),
- Rule-Based framework,
- Case-based framework,
- classifier-based framework,
- etc.

This section describes the ATN-based and rule-based frameworks.

2.1 ATN-Based Framework

The ATN-based framework relies on two fundamental notions: states and transitions which naturally build up an Augmented Transition Network (ATN). The main classes are therefore (see Figure ??): `ATN`, `State` and `Transition` (see package `Gdima.tools.automata`).

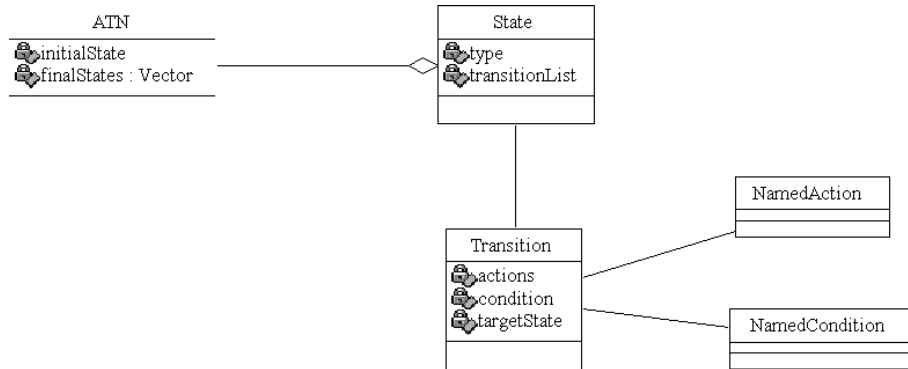


Figure 1: ATN-based framework classes

States represent decision points. They are used to choose the next transition among the associated transitions.

Each transition is labeled: condition, action (*if condition then* action). It represents a step of the global scheduling of the agent and links an input state with an output state.

The conditions of transitions test the occurrence of an asynchronous event (message reception, new data, ...) or an internal state change. These asynchronous events often operate changes on the context.

Figure 2 describes an Initiator of the contract protocol 8.

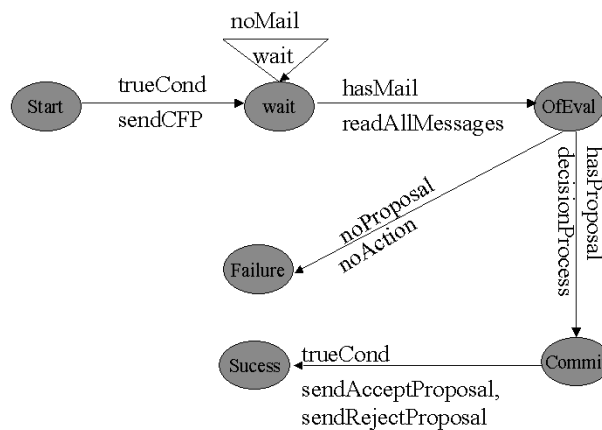


Figure 2: ATN of an Initiator

These actions and conditions are java methods that are implemented in a givencontext (Java object). For example, the action `sendCFP` of Figure 2 sends a call for proposal (CFP).

This framework provides also an ATN interpreter. The latter is mainly defined by a context which implements the conditions and actions. Conditions and actions instances of the classes `NamedCondition` and `NamedAction` respectively.

At each state, the ATN interpreter evaluates the conditions of transitions (representing new events) to select the most suited behavior. When these conditions are verified, the actions of the associated transition are executed and the ATN state is then modified.

```
public void run()
    State currentState = atn.getInitialState();
    while (!(currentState.isFinal()))
        currentState = currentState.crossTransition(context);
```

To define an ATN, one has to:

- define States (Instantiate `State`). There are three kinds of states: initial (one), Final (zero or more), intermediate (Zero or more).
- define Transitions. Each transition has conditions (one or more), actions (one or more) and a targetState.
- attach transition to States

The Contract Net initiator is described in the following: in the first case the ATN is described in a file and a parser (provided by the class `ATN`) is then used to instantiate ATN.

```
start
    trueCond ? sendCFP : wait

wait
    noMail ? wait : wait
    hasMail ? readAllMessages : OfEval

OfEval
    hasProposal ? descisionProcess : Commit
    noProposal ? noAction : Failure

Commit
    trueCond ? sendAcceptProposal, sendrejectProposal : Success
```

In the second case, it is described by the generic class `InitiatorInteractionModel`.

```

public InitiatorInteractionModel()
//Initial and Final States
State start = new State("start");
State failure = new State("failure"); failure.beFinal();
State success = new State("success"); success.beFinal();

//Other States
State wait = new State("wait");
State ofEval = new State("offerEvaluation");
State commit = new State("commitmentDecision");

// transitions between start and wait
start.setTransitions(new Transition("trueCond","sendCFP",wait));

// transitions from wait to wait and ofEval
wait.setTransitions(new Transition("noMail","wait",wait),
new Transition("hasMail","readAllMessages",ofEval));

// transitions from ofEval
ofEval.setTransitions(new Transition("hasProposal","decisionProcess",commit),
new Transition("noProposal","noAction",failure));

// transitions from commit
Vector va= new Vector (); va.add(new NamedAction("sendAcceptProposal"));
va.add(new NamedAction("sendRejectProposal"));
Vector vc = new Vector(); vc.add (new NamedCondition ("trueCond"));
commit.setTransitions(new Transition(vc,va,success));

//initialization of the role
Vector v = new Vector();
v.add(success);
v.add(failure);
this.setInitialState(start);
this.setFinalStates(v);
this.setName("ContractNetInitiator");

```

Note that several interaction models are provided as generic classes (see package `Gdima.kernel.communicatingAgent.InteractionProtocols`).

2.2 Rule-Based Framework

This framework defines an inference engine for small rule bases which don't require a compilation method such as RETE. It is provided by the package: `Gdima.tools.ruleBasedSystem` which has three Classes: `Rule`, `AbstractRuleBase`, `RuleBasedEngine`.

```

public class Rule extends RuleBasedObject
private java.util.Vector conditions; // list of conditions to match to trigger the rule
private java.util.Vector actions; // list of actions

```

An Object rule is defined by a set of conditions and actions that are similar to the conditions and action of an ATN (described in the section 2.1).

The class RuleBasedEngine implements the inference engine. It is mainly defined by a context which implements the conditions and actions.

3 A Generic Agent Architecture

An agent is an entity which has the following properties:

- An agent is an **autonomous** entity. It operates without direct intervention of humans or other agents. Therefore, it must have some control over its behaviors and its internal state [3].
- An agent is a **pro-active** entity. It has a goal and its activity is driven by this goal. It does not simply act in response to the received messages from the other agents.
- An agent is a **sociable** entity. For example, agents are considered to form groups, but a notion of *group* is rarely provided by active objects. Moreover, agents may interact by speaking different languages. For example, in the agent communication languages such as KQML (Knowledge Query and Manipulation Language) [5], the language of the message is indicated in an attribute of the received performative. This language can be an object-oriented language (Smalltalk, Java, ...) or an inter-change representation language (KIF, ...).
- An agent is an **adaptive** entity. Its world (environment and other agents) is continuously evolving. So, some behaviors must be adaptive to make the agent capable of sustained performance.

In an attempt to define a generic architecture which addresses the previous properties of an agent, several agent models have been proposed. Two main approaches can be distinguished: cognitive and reactive. In the cognitive approach, each agent contains a symbolic model of the outside world, about which it develops plans and makes decisions in the traditional (symbolic) AI way. In the reactive approach, on the other hand, simple-minded agents react rapidly to asynchronous events without using complex reasoning.

Neither a completely reactive nor a completely cognitive approach is suitable for building complete solutions for real-life applications. Hybrid models have been proposed to combine the advantages of both reactive and cognitive models. In these models, agents are decomposed in a set of modules which can in turn be of a reactive or cognitive nature. However, the design of hybrid agents is very complex and is more closer to cognitive agent than

reactive agent. It is not therefore easy to use these architectures to build simple agents. Moreover, they don't allow to have heterogeneous size of agents. So, they don't solve the problem of granularity. The solution of this problem is necessary for several real-life application.

In the following section, we present our generic and modular agent architecture.

3.1 A Modular Agent Architecture

DIMA agent architecture model is founded on the following conclusions: there are several interesting agent architectures. It is difficult to compare these agent architectures to find the best one. Each one is well suitable for a category of agents or application domains. A generic agent tool must thus integrate these architectures.

The agent architecture model of DIMA may be seen as an open model. A minimal agent kernel is proposed. This kernel can be then enriched, by incremental refinement, new functionalities can be added to this model. These functionalities are provided by the libraries of DIMA.

Modularity introduces more flexibility and allows to change easily components with the aim of improvement or tests. A modular agent architecture makes the agent an open system. Modularity provides thus several advantages to DIMA:

- Possibility to have variable granularity of agents.
- Possibility to have agents with adaptive structure. Each agent can dynamically change his components and the relations between these various components.
- Possibility to integrate different agent models.
- Possibility to include a library of reusable components
- Etc.

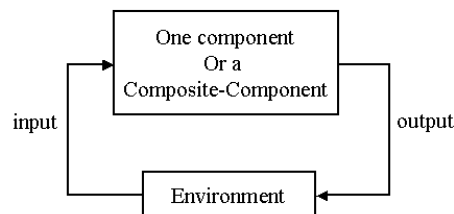


Figure 3: A Modular Agent Architecture

DIMA is therefore characterized by its modular architecture and by its various libraries. Each agent has one or more components (see Figure 3). For example, the communication module manages the interaction between the agent and some other agents of the system.

3.2 Agent Behavior

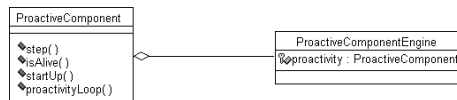


Figure 4: General structure of a ProactiveComponent

A Proactive component represents an autonomous and proactive entity. It provides the basic structure to represent behaviors. The kernel of DIMA is a framework of proactive components. It includes several classes and their methods. This framework is illustrated by a minimal set of classes and methods which define the main functionalities of a proactive component. These functionalities may be extended in the subclasses. This framework is mainly composed of the class `ProactiveComponent` (see Figure 4). An instance of `ProactiveComponent` describes:

- The goal of the proactive component, it is implicitly or explicitly described by the method `isAlive()`.
- the basic behaviors of the proactive component, a behavior is a sequence of actions that allow to change the internal state, to perform a message or to send a message to other components. Each behavior is implemented as a java method of this class.
- the metabehavior, it defines how the behaviors are selected, sequenced and activated. It relies on the goal (see the section 3.3 for details).

Table ?? describes the main methods of `ProactiveComponent`.

3.3 Agent Meta-Behavior

Many systems have emphasized the need for explicit and separate representation of control or the reflexive aspect of meta-level architectures. Following this tradition of explicit and separate representation of control, we propose a meta-behavior in our agent architecture. This meta-behavior gives each agent the ability to make appropriate decisions about control or to adapt its behaviors over time to new circumstances. It provides the agent with a self-control mechanism to dynamically schedule its behaviors in accordance to its internal state and its internal representation of its world.

This solution has the following foundations:

Table 1: Main methods of ProactiveComponent

Methods	Description
public abstract boolean isAlive()	Tests if the proactive component has not yet reach his goal.
public abstract void step()	represents a cycle of the meta-behavior of the proactive component.
void proactivityLoop()	Represents the meta-behavior of the proactive component. <pre>public void proactivityLoop() while (this.isAlive()) this.preActivity(); this.step(); this.postActivity();</pre>
public void startUp()	Initialize and activate the meta-behavior. <pre>public void startUp() this.proactivityInitialize(); this.proactivityLoop(); this.proactivityTerminate();</pre>

- behaviors are explicitly and dynamically scheduled by the meta-behavior,
- behaviors may be explicitly interrupted to allow the meta-behavior to adapt its scheduling according to new events.

The meta-behavior represents the agent self-control mechanism. This self-control provides the agent with some kind of control over its behaviors and its internal state. It defines the agent pro-activity which is not restricted to message reception/sending. Therefore, it makes the agent autonomous by allowing him to operate without direct intervention of humans or other agents. Moreover, it makes the agent adaptive by allowing him to deal rapidly with new events.

In `ProactiveComponent`, the meta-behavior is procedural. It is implicitly represented by the method `step()`. In this class, this method is abstract. Different paradigms (ATN, production rules,...) are then used to represent the agent meta-behavior. New subclasses of `ProactiveComponent` with these meta-behaviors have been therefore introduced. For instance, the meta-behavior of an `ATNBasedProactiveComponent` is modeled by an ATN. The states, in this case, correspond to the different states of the proactive component and actions correspond to the agent behaviors. The method

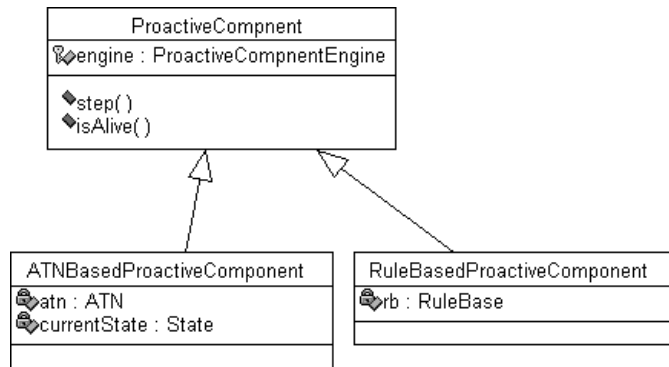


Figure 5: Hierarchy of Proactive Components

`step()` is in this case generic. It executes one step of the ATN Interpreter.

```
public void step()
    currentState = currentState.crossTransition(this);
```

Figure 5 gives the hierarchy of proactive components with examples of meta-behaviors.

3.4 Agent Engine

In *DIMA*, the class `ProactiveComponent` and its subclasses represent the internal activity of the agent. The instance method `proactivityLoop()` (see Table 1), used by `startUp`, defines the basic loop of the agents.

`AgentEngine` implements `Runnable`. In the latter, the method `run` has been redefined:

```
!AgentEngine methodsFor: 'activity setting'!
body
    self atnInterpreter
    public class AgentEngine extends ProactiveComponentEngine
    implements Runnable
    protected ProactiveComponent proactivity;
    public Thread thread;

public void run()
    proactivity.startUp();
```

The agent engine is therefore generic, it is not related to any application domain.

3.5 Communication and Interaction

To define communicating agents, we have reused the previous hierarchy of classes. The `BasicReactiveAgent` has been subclassed to define `ReactiveCommunicatingAgent` (see Figure 6). The latter introduces the basic communication behaviors: receive and send Messages. Each communicating agent must therefore have:

- a mail box to stock his messages.
- a communication component to manage the incoming and outgoing messages.

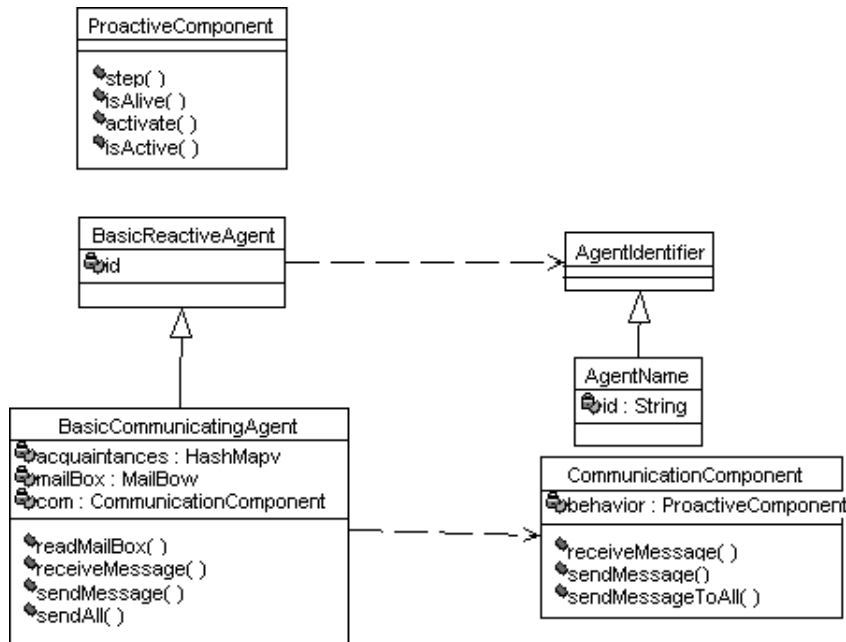


Figure 6: Hierarchy of Basic Agents

Each message is an instance of the class `Message` or its subclasses. The latter define the various types of KQML messages. For instance, the class `KQMLTell` implements KQML messages which have the performative `tell`.

Agent interaction protocol are often explicit. In this case, each agent may have one or several interaction models. In DIMA, these interaction models are implemented as subclasses of ATN. The set of actions and conditions of each interaction model are defined as Interface (see Figure 7). For instance, the interface `ContractNetInitiator` defines the methods of an Initiator. To use the Initiator interaction model, an agent class uses the class `InitiatorInteractionModel` to have the ATN and must implement the Interface `ContractNetInitiator`.

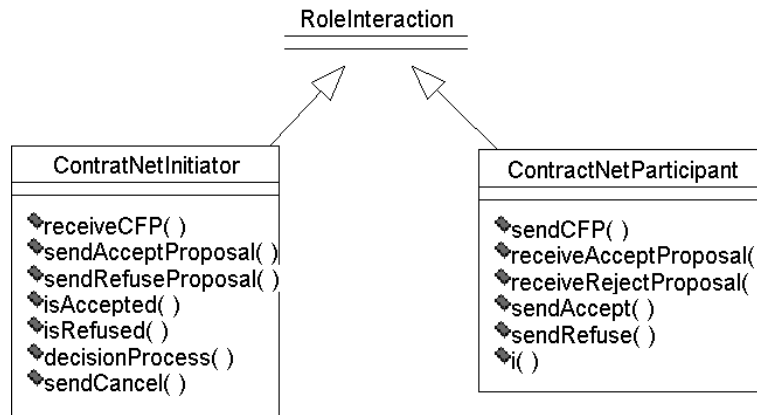


Figure 7: Hierarchy of Basic Agents

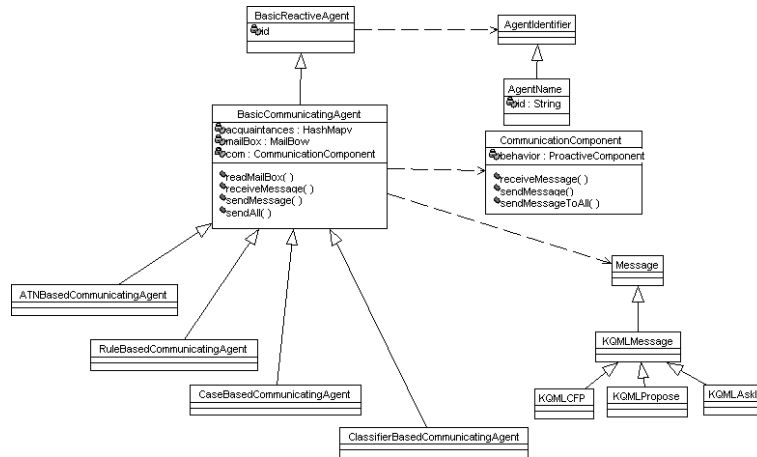


Figure 8: FIPA messages and basic agents

Moreover, DIMA is FIPA-Compliant. It provides therefore a hierarchy of classes which implement KQML messages, and the basic FIPA agents (see 8):

- Directory Facilitator (DF).
- Agent Management System (AMS).
- Agent Communication Channel (ACC).

4 Distribution and Replication

To distribute multi-agent system and to make them reliable, we developed DarX. DarX is a framework to design reliable distributed applications which

include a set of distributed communicating entities (agents). It is mainly used to develop fault-tolerant distributed multi-agent system.

DARX is built above Java, part of it using the RMI features, mainly because the multi-agent platforms DARX intends to support are implemented in this language.

To distribute a multi-agent system, one has to use the method: `activateWithDarx(activationURL, activationPORT)` instead of `activate`. All the examples can be therefore distributed easily without any problem.

More details on reliability may be found in [6].

5 Overview of DIMA

The use of an object-oriented language brings the benefit of the inheritance mechanism. DIMA provides a library of classes which can be easily used to implement multi-agent systems. These classes defines:

- agent behaviors,
- agent meta-behavior,
- agent communication components,
- FIPA agents,
- FIPA messages,
- etc.

Therefore, customizing *DIMA* means using or sub-classing the hierarchies of classes (see Figure 9).

6 Main Step to Develop a Multi-agent System

In *DIMA*, a multi-agent system is a set of agents and possibly a set of objects representing the agents domain. To implement a multi-agent system, one has :

1. to implement the domain, which is a collection of simple Java objects,
2. to implement of the agents by customizing *DIMA*. The main steps to implement an agent are the followings:
 - (a) Determination of the agent behaviors.
 - (b) Implementation of the agent class by using or sub-classing existing classes (see Figure 9).

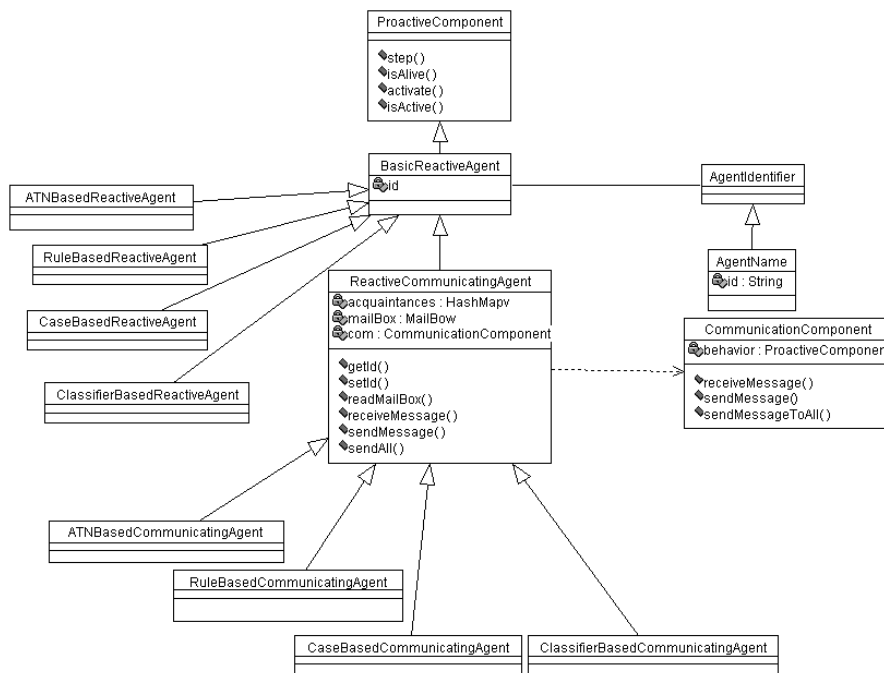


Figure 9: Examples of Classes of DIMA.

- (c) Implementation of the agent meta-behavior by instantiating an exiting class (e.g ATN) or introducing and then instantiating a new class.
 - (d) Instantiation of the agent class.
3. to initialize the agent acquaintances.
 4. to deploy and then to activate the agents.

7 Conclusion

Several architectures have been proposed (see [4] and [7]) to build agents out of two or more components which can be either cognitive or reactive. Reactive components are given some kind of precedence over the cognitive ones. A key problem in such architectures is what kind of control can be used to manage the interactions between these fundamentally different components.

DIMA proposes to decompose the behavior of an agent into an organization of behaviors and it uses a meta-behavior to control these different (reactive or cognitive) behaviors. It provides a library of classes to build easily agents.

We validated DIMA on several real-life applications:

- manufacturing process simulator;
- intensive care patient monitoring;
- modeling of firm and organizational form evolution;
- Information filtering;
- etc.

DIMA offers an interesting framework for studying multi-agent problems. For instance, to describe adaptive agents, we are currently studying an adaptive agent model and self-adaptive organization.

8 License

DIMA is free software and you are welcome to redistribute it under the terms of the GNU General Public License (version 2 or later versions). See the file `doc/licenses/GPL_license.txt`

In order to allow the development of commercial applications with DIMA, all classes defined in packages `GDIMA.xx.yy`, where `xx` and `yy` may be any name, are provided with a LGPL license.

This means that you may use DIMA freely and develop non-commercial applications based on GPL license without any trouble. However, when you want to develop commercial applications, you are restricted to the classes that are provided in the `DIMA.jar` libraries. Development tools, tutorials, etc. are restricted to non-commercial applications. Third party software and tools are provided with their own license, see below.

9 Third-party software

DIMA uses several third party libraries and software: JTP, Jena packages, etc. Each have their own license.

References

- [1] N. A. Avouris and L. Gasser, editors. *Distributed Artificial Intelligence*, chapter An Overview of DAI, pages 1–25. Kluwer Academic Publisher, Boston, 1992.
- [2] N. A. Avouris and L. Gasser. *Distributed Artificial Intelligence: Theory and Praxis*, chapter Object-Oriented Concurrent Programming and Distributed Artificial Intelligence, pages 81–108. Kluwer Academic Publisher, 1992.
- [3] C.Castelfranchi. A point missed in multi-agent, DAI and HCI. In M. J. Wooldridge and N. R. Jennings, editors, *Intelligent Agents - Theories, Architectures, and Languages*, number 890 in LNAI, pages 49–62. Springer Verlag, 1995.

- [4] I. A. Ferguson. *TouringMachines: An Architecture for Dynamic, Rational, Mobile Agents*. PhD thesis, University of Cambridge, Clare Hall, 1992.
- [5] T. Finin, R. Fritzson, D. McKay, and R. McEntire. KQML as an agent communication language. In *Third international conference on information and knowledge management*. ACM Press, November 1994.
- [6] Zahia Guessoum, Jean-Pierre Briot, and Sébastien Charpentier. Dynamic and adaptative replication for large-scale reliable multi-agent systems. In *Proceedings of the ICSE'02 First International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS'02)*, Orlando FL, U.S.A., may 2002. ACM.
- [7] J. Mller and M. Pischel. Modelling reactive behaviour in vertically layered agent architectures. In A. G. Cohen, editor, *Eleventh European Conference on Artificial Intelligence (ECAI'94)*, pages 709–713, Amsterdam, (NL), 1994.