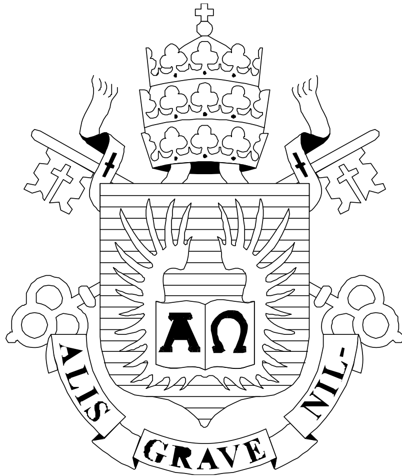


# PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 15/10

## BDI4JADE: a BDI layer on top of JADE

**Ingrid Oliveira de Nunes**
**Carlos José Pereira de Lucena**

Departamento de Informática

**PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO**
**RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22451-900**
**RIO DE JANEIRO - BRASIL**

Monografias em Ciência da Computação, No. 15/10 ISSN: 0103-9741
Editor: Prof. Carlos José Pereira de Lucena November, 2010

# BDI4JADE: a BDI layer on top of JADE [1]

**Ingrid Oliveira de Nunes and Carlos José Pereira de Lucena**

{ionunes,lucena}@inf.puc-rio.br

**Abstract.** Several agent platforms that implement the belief-desire-intention (BDI) architecture have been proposed. Even though most of them are implemented based on existing general purpose programming languages, such as the the Java language, the rely on a Domain-specific Language (DSL) writen in specific file types (e.g. XML). As a consequence, this prevents the integration with existing libraries and frameworks, and developers from using advanced language features. Due to these limitations of these BDI agent platforms, we have implemented the BDI4JADE, which is presented in this paper. It is a BDI layer on top of JADE, an agent platform.

**Keywords:** Multi-agent Systems, Agent Platforms, BDI Architecture, JADE.

**Resumo.** Muitas plataformas de agentes que implementam a arquitetura desejos-crenças-intenções (do inglês, BDI) foram propostas. Mesmo que a maior parte delas é implementada baseada em linguagens de programação de propósito geral existentes, tais como a linguagem Java, elas se baseiam em uma linguagem específica de domínio (do inglês, DSL) escrita em tipos de arquivo específicos (e.g. XML). Como conseqüência, isso não permite a intregração com bibliotecas e frameworks existentes e que desenvolvedores possam usar recursos avançados das linguagens. Devido a essas limitações destas plataformas de agentes BDI, nós implementamos o BDI4JADE, o qual é apresentado neste artigo. Ele uma camada BDI sobre o JADE, uma plataforma de agentes.

**Palavras-chave:** Sistemas Multi-agentes, Plataformas de Agente, Arquitetura BDI, JADE.

[1]This work has been partially supported by CNPq 557.128/2009-9 and FAPERJ E-26/170028/2008.

**In charge of Publications:**
Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC-Rio Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22451-900 Rio de Janeiro RJ Brasil
Tel. +55 21 3527-1516 Fax: +55 21 3527-1530
E-mail: bib-di@inf.puc-rio.br
Web site: http://bib-di.inf.puc-rio.br/techreports/

# Contents

1 Introduction 1

2 The belief-desire-intention (BDI) Model and Architecture 2

3 BDI4JADE: a BDI layer on top of JADE 3
3.1 BDI4JADE Core . . . . . 4
3.1.1 Reasoning Cycle . . . . . 5
3.1.2 Plan Selection . . . . . 8
3.1.3 Extension points . . . . . 8
3.2 Goals . . . . . 8
3.3 Beliefs . . . . . 9
3.4 Plans . . . . . 10
3.5 Messages . . . . . 11
3.6 Events . . . . . 12
3.7 Not implemented yet . . . . . 12

4 Related Work 13

5 Conclusion 13

References 14

# 1 Introduction

Agent-oriented Software Engineering (AOSE) (Jennings 1999, Wooldridge & Ciancarini 2000) is a software engineering paradigm in which complex systems can be decomposed into subsystems organized in a hierarchical way. These subsystems work together with the aim of achieving system goals as a whole. Moreover, each subsystem is an autonomous and proactive entity, each of which with its own goals and thread of control. Therefore, the agent abstraction becomes a natural metaphor to implement of this kind of system. A complex system is decomposed in terms of multiple autonomous components in a flexible manner, in order to achieve their goals.

It is not uncommon the need of incorporating cognitive abilities to agents. Besides their basic characteristics, namely autonomy, proactivity and social ability, agents can also have reasoning and learning abilities. Studies of cognitive science and artificial intelligence techniques are used to add these properties to agents. Software agents with a reasoning ability are denoted by cognitive agents.

One of the most widely used ways of designing and implementing cognitive agents is following the BDI model (Georgeff, Pell, Pollack, Tambe & Wooldridge 1999). The concepts of this model were initially proposed by Bratman (Bratman 1987). The model consists of beliefs, desires and intentions as mental attitudes, which generates human action. Rao & Georgeff presented in (Rao & Georgeff 1995) the BDI architecture. They adopted the model proposed by Bratman and transformed into in a formal theory and an execution model for software agents based on beliefs, goals and plans. This architecture served as a basis for the implementation of agent platforms. The first one that succeeded was the Procedural Reasoning Systems (PRS) (Georgeff & Lansky 1986).

Examples of agent platforms that implement the BDI architecture include JACK (Howden, Rönnquista, Hodgson & Lucas 2001, *JACK intelligent agents: JACK manual* 2005), Jason (Bordini, Wooldridge & Hübner 2007) and Jadex (Pokahr, Braubach & Lamersdorf 2003, Pokahr & Braubach 2007). In particular, these three platforms are based on the Java language. However, agents are implemented in these platforms in a Domain-specific Language (DSL) in specific file types (e.g. XML), which are processed and run in the Java platform. As a consequence, this prevents developers from using some features of the Java language, such as reflection and annotations, that help on the implementation of complex applications as well as integrate existing technologies. Due to these limitations of these BDI agent platforms, we have implemented a BDI layer on top of JADE, namely BDI4JADE, which is presented in this paper. JADE (Bellifemine, Claire & Greenwood 2007) is a Java-based agent platform that provides a robust infrastructure to implement agents, including behavior scheduling, communication and yellow pages service. We have leveraged these features provided by JADE, and built a BDI reasoning mechanism to JADE agents. Agents developed with our JADE extension are implemented in "pure" Java language, i.e. not in XML files.

The reminder of this paper is organized as follows. Section 2 briefly introduces the BDI model and architecture. Section 3 describes our BDI implementation on top of JADE. Section 4 discusses related work, and finally Section 5 presents final remarks.

## 2 The BDI Model and Architecture

There are several approaches that propose different types of mental attitudes and their relationships. Among them, the most adopted is the belief-desire-intention (BDI) model, originally proposed by Bratman (Bratman 1987) as a philosophical theory of the practical reasoning, explaining the human reasoning with the following attitudes: beliefs, desires and intentions. The essential assumption of the BDI model is that actions are derived from a process named practical reasoning, which is composed of two steps. In the first step, deliberation (of goals), a set of desires is selected to be achieved, according to the current situation of the agent's beliefs. The second step is responsible for the determination of how these concrete goals produced as a result of the previous step can be achieved by means of the available options for the agent (Wooldridge 2000).

The three mental attitudes that are part of the BDI model are described next.

**Beliefs.** They represent environment characteristics, which are updated accordingly after the perception of each action. They can be seen as the informative component of the system.

**Desires.** They store the information of the goals to be achieved, as well as properties and costs associated with each goal. They represent the motivational state of the system.

**Intentions.** They represent the current action plan chosen. They capture the deliberative component of the system.

Rao& Georgeff (Rao & Georgeff 1995) adopted the BDI model for software agents and presented a formal theory and an abstract BDI interpreter, which is the base for almost all BDI systems, either historical or used at the present. The interpreter operates over beliefs, goals and plans of the agent, which represent the concepts of the mentalistic notions, with small modifications. The most significant change is that goals are a set of consistent concrete desires that can be achieved all together, avoiding the need of a complex phase of goal deliberation. The main task of the interpreter is the realization of the means-end process by means of the selection and execution of plans for a certain goal or event. The first system implemented with success based on this interpreter was the Procedural Reasoning Systems (PRS) (Georgeff & Lansky 1986), which has as a successor the system named dMARS (d´Inverno, Kinny, Luck & Wooldridge 1997, D´Inverno, Luck, Georgeff, Kinny & Wooldridge 2004).

The process of practical reasoning in a BDI agent is presented in Figure 1 (source (Wooldridge 1999)). As shown in this figure, there are seven main components in a BDI agent:

- a set of current **beliefs**, representing information the agent has about its current environment;
- a **belief revision function**, which takes a perceptual input and the agent's current beliefs, and on the basis of these, determines a new set of beliefs;
- an **option generation function**, (options), which determines the options available to the agent (its desires), on the basis of its current beliefs about its environment and its current intentions;

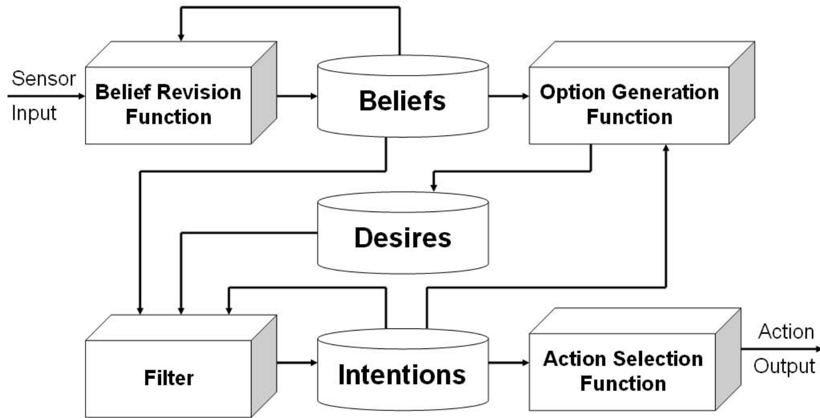


Figure 1: A generic diagram of the BDI architecture.

- a set of current **options**, representing possible courses of actions available to the agent;
- a **filter function** (filter), which represents the agent's deliberation process, and which determines the agent's intentions on the basis of its current beliefs, desires, and intentions;
- a set of current **intentions**, representing the agent's current focus – those states of affairs that it has committed to trying to bring about;
- an **action selection function**, which determines an action to perform on the basis of current intentions.

## 3 BDI4JADE: a BDI layer on top of JADE

As stated in the introduction, our motivation for implementing a new BDI platform is that the languages, even though based on general purposed programming languages, provided by existing platforms limit the integration with up-to-date available technologies, such as frameworks and libraries and also the use of advanced features of the underlying programming language.

One agent framework that meets our motivation is JADE. Nevertheless, it is not based on the BDI model. Instead, it implements a task-oriented model, in which agents have a set of behaviors. No cognitive abilities are provided for agents. However, JADE is a robust and mature infrastructure, and provides many features that are needed for implementing Multi-agent Systems (MASs), which include the yellow pages service and messages exchange. In addition, the behaviors scheduler can be leveraged to control the execution of plans in BDI agents. So, instead of developing an agent platform from scratch, we added a layer on top of JADE, which implements the BDI architecture. This layer is developed in "pure" Java, i.e. no XML files, what makes it easy to integrate with existing applications and reusable assets (frameworks, components, libraries). These are some of the main characteristics of our JADE extension:

- *Use of capabilities.* Agents aggregate a set of capabilities, which are a collection of beliefs and plans. This allows modularizing particular agents' behaviors.
- *Java generics for beliefs.* Beliefs can store any kind of information and are associated with a name. If the value of a belief is retrieved, it must be cast to its specific type. We have used Java generics to capture incorrect castings in compile time.
- *Plan bodies are instance of JADE behaviors.* In order to better exploit JADE features, in particular its behaviors hierarchy, plan bodies in our extension are subclasses of JADE behaviors. To make this possible, they have to implement the `PlanBody` interface of our extension as well.

Next, we detail our JADE extension, the BDI4JADE. We first present the core of our implementation, which consists of agents, intentions, capabilities and the reasoning cycle, and its whole structure. Then, we describe individual BDI4JADE components – how they were implemented and how to extend them. Finally, we point out some parts of the framework that have not been implemented yet. However, they do not interfere in how our implementation works, they only will improve it.

### 3.1 BDI4JADE Core

A BDI agent in our platform must extend the `BDIAgent` class, which in turn is an extension of the `Agent` class from JADE. A `BDIAgent` (from now on, we refer it to as agent) is composed of a set of intentions (`Intention` class) and a set of capabilities (`Capability` class).

When a goal is added to an agent, a new associated intention is created and added to it. Intentions have a status associated with it, which are: (i) ACHIEVED – the goal associated with that intention was achieved; (ii) NO_LONGER_DESIRED – the goal associated with that intention is no longer desired; (iii) PLAN_FAILED – the agent is trying to achieve the goal associated with that intention, but the last executed plan has failed; (iv) TRYING_TO_ACHIEVE – the agent is trying to achieve the goal associated with that intention, but it is executing a plan for achieving it; (v) UNACHIEVABLE – all available plans were executed to try to achieve the goal associated with that intention, but none of them succeeded; and (vi) WAITING – the agent has the goal, but it is not trying to achieve it.

In the BDI model, an intention is a goal that an agent is committed to achieve. Our implementation does not make this distinction explicitly, but implicitly. A goal from the BDI model is an intention from the BDI4JADE with the status WAITING. An intention from the BDI model is an intention from the BDI4JADE with the status PLAN_FAILED and TRYING_TO_ACHIEVE. Finally, an intention from the BDI4JADE with the status ACHIEVED, NO_LONGER_DESIRED, UNACHIEVABLE were intentions of the BDI model. They are in a final state to be removed from the agent.

Beliefs and plans are not part from an agent (directly), as proposed in the BDI model, but part of capabilities. A capability (Busetta, Howden, Rönnquist & Hodgson 2000) is essentially a set of plans, a fragment of the knowledge base that is manipulated by those plans and a specification of the interface to the capability. This concept is implemented by JACK and Jadex agent platforms. Capabilities have been introduced into some MASs as a Software Engineering (SE) mechanism to support modularity and reusability while still

allowing meta-level reasoning. As opposed to JACK and Jadex, beliefs and plans in our platform are not part of capabilities AND agents, but only capabilities. However, a belief, or a plan, can be part of an agent if all capabilities contain that belief, or that plan. As we deal with Java objects, this can be easily done, because all capabilities will have a pointer for the same object.

A capability from our framework is essentially composed of a belief base (BeliefBase class) and plan library (PlanLibrary class). The first is a collection of beliefs (see Section 3.3), and the latter a collection of plans (see Section 3.4).

All these components – capability, belief base and plan library – can be implemented either by extension or instantiation. A developer can extend these components in the code and override the empty implementations of the setup() method for capabilities and the init() method for belief bases and plan libraries to initialize these components. The other option is to instantiate these components and add beliefs and plans by method invocation.

As opposed to typical BDI platforms, ours does not have an explicit declaration of goals in agents and capabilities. This binding occurs only at runtime. This provides more flexibility, because plans can be added (learned) to plan libraries at runtime and then goals (which could be unknown at development time) can be added (desired) and achieved at runtime.

Figure 2 depicts the class model of the BDI4JADE, presenting all these discussed components as well as the ones that are going to be described in next sections.

### 3.1.1 Reasoning Cycle

An essential part of a BDI agent platform is the reasoning cycle that it provides to be part of agents. Listing 1 shows the source code of the reasoning cycle implemented in our platform.

The first step (line 2) corresponds to the belief revision function introduced in Section 2. It is performed by invoking the method void reviewBeliefs(BDIAgent) from a BeliefRevisionStrategy. Next (lines 6-21), all finished intentions, i.e. intentions whose status is ACHIEVE, NO_LONG_DESIRED or UNACHIEVABLE, are removed from the set of intentions of the agent, and a map goalStatus is created to store the status of each current goal of the agent.

Then (lines 23-24), the method Set<Goal> generateGoals(Map<Goal, GoalStatus>) from an instance of OptionGenerationFunction is invoked to creating new goals or to dropping existing ones. This is associated with the option generation function in Figure 1. Based on the set of goals received as output, two actions are performed: (i) **new** goals are added to the agent, and consequently associated intentions are created (lines 25-29); and (ii) **removed** goals are set as no longer desired and removed from the agent (lines 30-42). Existing goals, but not removed, remain unchanged. The goalStatus is then updated (lines 44-47).

Next, it is time of the deliberation process (filter in Figure 1). This is performed by invoking the method Set<Goal> filter(Map<Goal, GoalStatus>) from the class DeliberationFunction (line 48). It selects a set of goals that must be tried to achieve (intentions) from the set of goals. Selected goals, and associated intention, will be set to trying to achieve, and unselected goals, and associated intentions will be set to a waiting state. The invocation of the methods in line 51 and 53 correctly adjusts the new state of the intention.

Figure 2: BDI4JADE class model.

Listing 1: BDI4JADE Reasoning Cycle.

```
public void action() {
    beliefRevisionStrategy.reviewBeliefs(BDIAgent.this);

    synchronized (intentions) {
        Map<Goal, GoalStatus> goalStatus = new HashMap<Goal, GoalStatus>();
        Iterator<Intention> it = intentions.iterator();
        while (it.hasNext()) {
            Intention intention = it.next();
            GoalStatus status = intention.getStatus();
            switch (status) {
            case ACHIEVED:
            case NO_LONGER_DESIRED:
            case UNACHIEVABLE:
                intention.fireGoalFinishedEvent();
                it.remove();
                break;
            default:
                goalStatus.put(intention.getGoal(), status);
                break;
            }
        }

        Set<Goal> generatedGoals = optionGenerationFunction
                .generateGoals(goalStatus);
        Set<Goal> newGoals = new HashSet<Goal>(generatedGoals);
        newGoals.removeAll(goalStatus.keySet());
        for (Goal goal : newGoals) {
            addGoal(goal);
        }
        Set<Goal> removedGoals = new HashSet<Goal>(goalStatus.keySet());
        removedGoals.removeAll(generatedGoals);
        for (Goal goal : removedGoals) {
            it = intentions.iterator();
            while (it.hasNext()) {
                Intention intention = it.next();
                if (intention.getGoal().equals(goal)) {
                    intention.noLongerDesire();
                    intention.fireGoalFinishedEvent();
                    it.remove();
                }
            }
        }

        goalStatus = new HashMap<Goal, GoalStatus>();
        for (Intention intention : intentions) {
            goalStatus.put(intention.getGoal(), intention.getStatus());
        }
        Set<Goal> selectedGoals = deliberationFunction.filter(goalStatus);
        for (Intention intention : intentions) {
            if (selectedGoals.contains(intention.getGoal())) {
                intention.tryToAchive();
            } else {
                intention.doWait();
            }
        }

        if (intentions.isEmpty()) {
            this.block();
        }
    }
}
```

This reasoning cycle is implemented as a `CyclicBehaviour` of JADE, therefore it is performed continuously, in addition, it is added to all instances of `BDIAgent`. The if condition in line 57 tests if the agent has no current intentions, and, if so, it blocks the behavior. This avoids that this behavior is continuously executed while there are no intentions. In case a new intention is added to the agent, the reasoning cycle is resumed.

### 3.1.2 Plan Selection

When the intention status is set to TRYING_TO_ACHIEVE or PLAN_FAILED, the private method `void dispatchPlan()` of the `Intention` class is invoked in order to select and execute a plan to try to achieve the goal associated with the intention.

This method first retrieves all plans that can achieve the goal, and then removes from this set of plans all plans that were already executed. The set of all plans that can achieve the goal is generated each time the `dispatchPlan()` method is executed because while a previous plan was being executed, new plans can be added to any capability of the agent. If there is no plan that can achieve the goal, the intention is set to UNACHIEVABLE. Otherwise, a plan will be selected by invoking the method `Plan selectPlan(Goal goal, Set<Plan>)` from the strategy `PlanSelectionStrategy` of the agent. After the plan selection, it will be instantiated and started.

### 3.1.3 Extension points

We have mentioned above four strategies: `BeliefRevisionStrategy`, `OptionGenerationFunction`, `DeliberationFunction` and `PlanSelectionStrategy`, and have not given too much details about it. These are Java interfaces, and are extension points of our platform.

Developers can customize a `BDIAgent` by setting the implementation to be used during the reasoning cycle of a specific agent. BDI4JADE provides a default implementation for each of these strategies:

- `DefaultBeliefRevisionStrategy` – it invokes the `BeliefBase.reviewBeliefs()` for the belief base of all capabilities;
- `DefaultOptionGenerationFunction` – it returns the current set of goals, i.e. do not drop anyone and do not create any new goal;
- `DefaultDeliberationFunction` – it returns the whole set of goal, i.e. all goals will go to a trying to achieve state; and
- `DefaultPlanSelectionStrategy` – it returns null if the set of plans is empty, and the first plan retrieved from the set, otherwise.

This way of extending and customizing agents is an implementation of the strategy design pattern (Gamma, Helm, Johnson & Vlissides 1995).

## 3.2 Goals

A goal in BDI4JADE can be any Java object, with the condition that it must implement `Goal` interface. Therefore, a class implementing this interface can be created and attributes can be added to by inputs and outputs of the goal.

We also provide a set of goals to be used in applications:

`BeliefGoal`. The input of this goal is the name of a belief. This goal is achieved when a belief with the provided name is part of the agents' beliefs.

`BeliefSetValueGoal<T>`. The input of this goal is the name of a belief and a value. This goal is achieved when the belief with the provided name is part of the agents' beliefs and has the provided value.

`CompositeGoal`. This class represents an abstract goal that is a composition of other goals (subgoals). It has two subclasses, which indicate if the goals must be achieved in a parallel or sequential way.

`ParallelGoal`. This class represents a goal that aims at achieving all goals that compose it in a parallel way.

`SequentialGoal`. This class represents a goal that aims at achieving all goals that compose it in a sequential way.

`MessageGoal`. This goal is created when a message is received by the agent. It stores the message received. How this goal will be achieved is described in Section 3.5.

In order to add a new goal to an agent, the only thing that must be done is to invoke the method `void addGoal(Goal goal)` from the `BDIAgent`.

## 3.3 Beliefs

Beliefs in the BDI4JADE are instance of the class `Belief<T>` or any of its subclasses. A belief has to main properties: a name and a value. The belief name must be unique in the scope of a belief base. There are two main characteristics about beliefs to be described: (i) its class is generic, i.e. it receives a type when it is instantiated. Therefore, when a belief is declared in a plan or somewhere else, no type casting must be performed to retrieve its value; and (ii) it extends the class `MetadataElement`, which is a class of metadata – a map from string to objects. Metadata can be used for specific purposes of applications, for instance, time can be added to beliefs, so they can be forgot after a certain amount of time.

The `Belief<T>` is an abstract class, because it does not specify how the value is stored, but defines methods that must be implemented by subclasses to retrieve and set the value associated with the belief. Currently, there is only one form of storing beliefs, which is implemented by the `TransientBelief<T>` class. This class stores the value of the type `T` in memory, and there is no persistence mechanism.

In addition, there is a particular type of belief to store sets – the `BeliefSet<T>`, which extends `Belief<Set<T>>`. As the `Belief<T>` class, it is abstract and can have different subclasses of to store the belief values. The `BeliefSet<T>` defines methods to retrieve, store and iterate the belief values, and has an implementation that stores value in memory – the `TransientBeliefSet<T>` class.

The `BeliefBase` class offers methods to manipulate beliefs, such as add, remove and update beliefs.

### 3.4 Plans

Plans in the BDI4JADE are related to a set of classes. One of the reasons is that our goal is to reuse as much as possible of JADE. First, to facilitate the learning process of developers already familiar with JADE; second, to exploit reuse benefits – which is higher quality due to the use of a piece of software used a lot of times, and reduced development costs. Plans to be executed (plan bodies) in our platform are instances of the JADE behavior.

Our platform has three main classes associated with plans:

`Plan`. A plan does not state a set of actions to be executed in order to achieve a goal, but have some information about it, which are: (i) the plan id; (ii) the plan library that is belongs to; (iii) the goals that it is able to achieve; and (iv) the message templates it can process. In addition, it defines some important methods to be implemented by subclasses.

- `public abstract Behaviour createPlanBody()` – this method returns an instance of a JADE behavior, which corresponds to the body to be executed to achieve the goal. This behavior instance must also implement the `PlanBody` interface.
- `protected void initGoals()` – this method must be overridden by subclasses to initiate the set of goals that this plan can achieve.
- `protected void initMessageTemplates()` – this method must be overridden by subclasses to initiate the set of message templates (from JADE) that this plan can achieve.
- `protected boolean matchesContext(Goal goal)` – this method verifies a context to determine if the plan can achieve the goal according to the current situation of the environment. The default implementation returns always `true`.

The first method must be implemented, because it is an abstract method, and therefore the `Plan` class is also abstract.

The method presented in Listing 2 is executed to verify if a plan can achieve a given goal.

Our platform provides a concrete implementation of `Plan`, the `SimplePlan`. This class has a `Class<? extends Behaviour>` associated with it, which must also implement the `PlanBody` interface (test made at runtime). When the `createPlanBody()` is invoked, in instance of the class associated with the `SimplePlan` will be created. This class in turn has two subclasses used to achieve generically sequential and paralell goals (see Section 3.2).

`PlanInstance`. This class, as the name indicates, is an instance of a plan. It is created to achieve a particular goal, according to a specification of a plan. It has the following attributes: (i) `Behaviour behaviour` – the behavior being executed to achieve the goal associated with the intention; (ii) `Intention intention` – the intention whose goal is trying to be achieved; (iii) `Plan plan` – the plan that this plan instance is associated with; (iv) `EndState endState` – the end state of the plan instance (FAILED or SUCCESSFUL), or `null` if it is currently being executed; (v) `List<Goal> subgoals` – the subgoals dispatched by this plan. In case of the goal of the intention associated

with this plan of this plan instance is dropped, all subgoals are also dropped; and (iv) `List<GoalFinishedEvent> goalEventQueue` – when this plan instance dispatches a goal, it can be notified when the dispatched goal finished (achieved, considered unachievable, and so on).

`PlanBody`. As we established that JADE behaviors would be used to execute plans and that we aimed at reusing the JADE behaviors hierarchy, we could not extend the `Behaviour` class of JADE. So, our decision was to define an interface that plan bodies should implement, besides extending a JADE behavior. Two methods should be implemented by plan bodies: (i) `EndState getEndState()` - it returns the end state of the plan body. If it has not finished yet, it should return `null`; and (ii) `void init(PlanInstance planInstance)` - this method is invoked when the plan body is instantiated. This is used to initialize it, for instance retrieving parameters of the goal to be achieved.

Listing 2: Verifying if a plan can achieve a goal.

```
1 public boolean canAchieve(Goal goal) {
2     if (goal instanceof MessageGoal) {
3         return canProcess(((MessageGoal) goal).getMessage());
4     } else {
5         return goals.contains(goal.getClass()) ? matchesContext(goal)
6                 : false;
7     }
8 }
```

In order to dispatch a goal and wait for its end, we adopted a mechanism similar to the one of receiving messages in JADE. The developer, after dispatching the goal, should retrieve a goal event and test if it is `null` (no goal event received yet) or not (an event was received). Listing 3 shows an example of how it could be done. The method `dispatchSubgoalAndListen()` blocks the behavior in case there is no goal event when it was invoked (a timeout can be provided for the method). The behavior will become active again when a goal event is received.

### 3.5 Messages

Messages are received and sent in the BDI4JADE basically as it is done in JADE. Conversations are made by sending messages, and then using the method `receive(MessageTemplate)` to receive a reply. Additionally, BDI4JADE provides an additional mechanism for processing messages that are received.

Every `BDIAgent` has a behavior `BDIAgentMsgReceiver` associated with it, which extends the `MsgReceiver` class from JADE. The latter is a behavior that handles a message when the match expression of the behavior returns a `true` value related to the analysis of the message received. The match expression of the `BDIAgentMsgReceiver` class checks if any of the capabilities of the agent have at least one plan that can process the received message. If so, the expression returns `true`. After that, the behavior adds a `MessageGoal` to the agent, with the received message associated with it. Eventually, the reasoning cycle will select a plan to process the message.

Listing 3: Dispatching and waiting for subgoals.

```
1  (...)
2  switch (state) {
3      case 0:
4          planInstance.dispatchSubgoalAndListen(subgoal);
5          state++;
6          break;
7      case 1:
8          GoalFinishedEvent goalEvent = planInstance.getGoalEvent();
9          if (goalEvent == null) {
10             return;
11         } else {
12             if (GoalStatus.ACHIEVED.equals(goalEvent.getStatus())) {
13                 (...)
14             } else {
15                 (...)
16             }
17         }
18         break;
19 }
20 (...)
```

### 3.6 Events

Our platform implements the observer design pattern (Gamma et al. 1995) in some points in order to enable the observation of events that occur in an agent. Currently, there are two kinds of events: belief and goal events.

Belief listeners can be associated with a belief base, and whenever a belief is added, removed or changed, the listener will be notified. It is important to highlight that a belief can have its value changed simple by invoking the `void setValue(T)` method, and in this case, the listeners will not be notified.

Goal listeners in turn are associated with an intention. It is used to observe changes in the status of the intention. And example of use was presented in Section 3.4, in order to detect when a subgoal was achieved (or finished in another state).

### 3.7 Not implemented yet

The components described in previous sections are already implemented and working. However, there are some assets present in the source code of BDI4JADE (Nunes 2010) that are place holders for future extensions of the platform. They are:

1. **Persistent beliefs.** Currently, our platform only provides the creating of transient beliefs. We intend to incorporate the Hibernate[2] framework to our platform to facilitate the creation of beliefs that are persisted in databases.

2. **Control of intention/goal owners.** We have created the interface `InternalGoal` to denote a goal that is internal to a capability. Plans that are being executed are associated with a plan library, which is in turn associated with a capability. Therefore, if the plan dispatches a goal, this goal is under the scope of this capability.

[2]http://www.hibernate.org/

This information is not being currently stored. Our goal is to limit the scope of the searching space of plans to the capability that dispatched the goal, when the goal is an `InternalGoal`. This helps creating encapsulated capabilities and improving reuse.

3. **Indexes in plan libraries.** Every time a plan must be selected for achieving a goal, the plan library is asked to provide the list of plans that can achieve that goal. We aim at creating indexes for speeding up this process.

# 4 Related Work

We described previously that other BDI agent platforms require developers to implement agents in DSLs. Jason (Bordini et al. 2007) agents are implemented in an extension of the AgentSpeak language (Rao 1996), which are written in ".asl" files. JACK (Howden et al. 2001, *JACK intelligent agents: JACK manual* 2005) has an specific language, the JACK Agent Language, which are compiled for Java.

The framework that has more similarities to BDI4JADE is Jadex (Pokahr et al. 2003, Pokahr & Braubach 2007), which is also a layer on top of JADE. Our experience with the development of several applications using Jadex was also a motivation for developing a new layer on top of JADE, which is not based on XML, as it is in Jadex. The experience with the development of a case study was reported in (Nunes, Cirilo & Lucena 2009).

The main benefit of Jadex is that it provides the concepts of the BDI architecture for developers, therefore an agent modeled using this architecture may be directly implemented without defining an implementation strategy for the modeled concepts. In addition, the capability concept is very useful to modularize parts of the agent. As a consequence, one can easily (un)plug capabilities from agents and reuse them.

However, Jadex defines agents into XML files, and this brings drawbacks during the implementation. Finding errors in XML files is a tedious task. Additionally errors are not captured during compilation time, because typos may occur even though the document is valid according to its DTD. For instance, if a goal is referenced within the XML file with a wrong letter, an error will occur only during execution time, and the message is that the XML file has errors. As a consequence, the developer has to find the error manually. Moreover, even though plans are Java classes, beliefs and parameters are retrieved by methods that return an object of the class `Object`, so there must be type casting while invoking these methods. This leads again to capturing errors only at runtime. Furthermore, the use of XML files is not appropriate for adopting modularization techniques, as discussed above.

# 5 Conclusion

Several BDI platforms have been proposed, and they mostly focused on implementing the BDI architecture – its abstractions and reasoning cycle. However, they have limitations related with the integration of existing technologies, such as framework and libraries, and the use of features of the underlying general purpose programming language.

In this paper we presented BDI4JADE, an agent platform that implements the BDI architecture. As the implementation of agents in this platform is performed only in Java

(not XML code or other kind of configuration files), it facilitates the integration with other technologies. BDI4JADE is a BDI layer on top of JADE, and it leverages all the features provided by the framework. Our platform as well as examples of its use are available in (Nunes 2010). BDI4JADE is being used in the context of our current research work (Nunes, Barbosa & Lucena 2010*a*, Nunes, Barbosa & Lucena 2010*b*).

## References


Bellifemine, F. L., Claire, G. & Greenwood, D. (2007), *Developing Multi-Agent Systems with JADE*, John Wiley & Sons, Inc., New York, USA.

Bordini, R. H., Wooldridge, M. & Hübner, J. F. (2007), *Programming Multi-Agent Systems in AgentSpeak using Jason (Wiley Series in Agent Technology)*, John Wiley & Sons.

Bratman, M. E. (1987), *Intention, Plans, and Practical Reason*, Cambridge, MA.

Busetta, P., Howden, N., Rönnquist, R. & Hodgson, A. (2000), Structuring bdi agents in functional clusters, *in* 'ATAL '99', pp. 277–289.

d´Inverno, M., Kinny, D., Luck, M. & Wooldridge, M. (1997), A formal specification of dMARS, *in* 'Agent Theories, Architectures, and Languages', pp. 155–176.

D´Inverno, M., Luck, M., Georgeff, M. P., Kinny, D. & Wooldridge, M. J. (2004), 'The dMARS architecture: A specification of the distributed multi-agent reasoning system', *Autonomous Agents & Multi-Agent Systems* **9**, 5–53.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995), *Design Patterns: Elements of Reusable Object-oriented Software*, Addison-Wesley.

Georgeff, M. & Lansky, A. (1986), Procedural knowledge, *in* 'IEEE Special Issue on Knowledge Representation', Vol. 74, pp. 1383–1398.

Georgeff, M., Pell, B., Pollack, M., Tambe, M. & Wooldridge, M. (1999), The belief-desire-intention model of agency, *in* J. Müller, M. P. Singh & A. S. Rao, eds, 'Proceedings of the 5th International Workshop on Intelligent Agents V : Agent Theories, Architectures, and Languages (ATAL-98)', Vol. 1555, Springer-Verlag: Heidelberg, Germany, pp. 1–10.

Howden, N., Rönnquista, R., Hodgson, A. & Lucas, A. (2001), Jack intelligent agents™: Summary of an agent infrastructure, *in* 'The Fifth International Conference on Autonomous Agents', Montreal, Canada.

*JACK intelligent agents: JACK manual* (2005), Technical Report 4.1, Agent Oriented Software Pvt. Ltd, Melbourne, Australia.

Jennings, N. R. (1999), Agent-Oriented Software Engineering, *in* F. J. Garijo & M. Boman, eds, 'Proceedings of the 9th European Workshop on Modelling Autonomous Agents in a Multi-Agent World : Multi-Agent System Engineering (MAAMAW-99)', Vol. 1647, Springer-Verlag: Heidelberg, Germany, pp. 1–7.

Nunes, I. (2010), 'A bdi extension for jade'. http://www.inf.puc-rio.br/˜ionunes/bdi4jade/.

Nunes, I., Barbosa, S. & Lucena, C. (2010*a*), An end-user domain-specific model to drive dynamic user agents adaptations, *in* 'SEKE 2010', USA.

Nunes, I., Barbosa, S. & Lucena, C. (2010*b*), Increasing users' trust on personal assistance software using a domain-neutral high-level user model, *in* T. Margaria & B. Steffen, eds, 'Leveraging Applications of Formal Methods, Verification, and Validation (ISoLA 2010)', Vol. 6415 of *LNCS*, Springer, Heraclion, Greece, pp. 473–487.

Nunes, I., Cirilo, E. & Lucena, C. (2009), Developing a family of software agents with fine-grained variability: an exploratory study, *in* 'V Workshop on Software Engineering for Agent-oriented Systems (SEAS 2009)', Fortaleza, Brazil, pp. 71–82.

Pokahr, A. & Braubach, L. (2007), Jadex user guide, Technical Report 0.96, University of Hamburg, Hamburg, Alemanha.

Pokahr, A., Braubach, L. & Lamersdorf, W. (2003), 'Jadex: Implementing a BDI-Infrastructure for JADE Agents', *EXP – in search of innovation* **3**(3), 76–85.

Rao, A. S. (1996), Agentspeak(l): Bdi agents speak out in a logical computable language, *in* 'MAAMAW '96: Proceedings of the 7th European workshop on Modelling autonomous agents in a multi-agent world : agents breaking away', Springer-Verlag New York, Inc., Secaucus, NJ, USA, pp. 42–55.

Rao, A. S. & Georgeff, M. P. (1995), BDI-agents: from theory to practice, *in* 'Proceedings of the First Intl. Conference on Multiagent Systems', San Francisco.

Wooldridge, M. (1999), Intelligent agents, *in* 'Multiagent systems: a modern approach to distributed artificial intelligence', MIT Press, Cambridge, MA, USA, pp. 27–77.

Wooldridge, M. & Ciancarini, P. (2000), Agent-Oriented Software Engineering: The State of the Art, *in* P. Ciancarini & M. Wooldridge, eds, 'First Int. Workshop on Agent-Oriented Software Engineering', Vol. 1957, Springer-Verlag, Berlin, pp. 1–28.

Wooldridge, M. J. (2000), *Reasoning about Rational Agents*, MIT Press.