BDI4JADE: a BDI layer on top of JADE

Ingrid Nunes[1,2], Carlos J.P. de Lucena[1], and Michael Luck[2]

[1] PUC-Rio, Computer Science Department, LES - Rio de Janeiro, Brazil
{ionunes,lucena}@inf.puc-rio.br
[2] King's College London, Strand, London, WC2R 2LS, United Kingdom
michael.luck@kcl.ac.uk

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, e.g. Java, agents are either programmed in a new programming language or Domain-specific Language expressed in XML. As a consequence, this prevents the use of advanced features of the underlying programming language and the integration with existing libraries and frameworks, which are essential for the development of enterprise applications. Due to these limitations of BDI agent platforms, we have implemented the BDI4JADE, which is presented in this paper. It is implemented as a BDI layer on top of JADE, a well accepted agent platform.

Keywords: Multi-agent Systems, Agent Platforms, Agent Programming, BDI Architecture, BDI4JADE, JADE.

1 Introduction

With the popularity of the web, complex systems has become a reality. These are characterized by being distributed and composed of multiple autonomous entities, which interact with each other. Multi-agent systems are considered a promising approach for developing this kind of systems [17], by decomposing them into agents, each of which with its own thread of execution, possibly a proactive behavior, thus aiming to achieve its individual goals, and able to perceive its surrounding environment and respond in a timely fashion to changes to it. From a software engineering perspective, multi-agent systems can be seen as a paradigm in which systems are decomposed into autonomous and proactive software components, namely agents.

Due to the complexity associated with the development of multi-agent systems, which typically involves thread control, message exchange across the network, cognitive ability, and discovery of agents and their services, several architectures and platforms have been proposed. One of the widely known architectures for designing and implementing cognitive agents is the belief-desire-intention (BDI) architecture, following a model initially proposed by Bratman [3], which consists of beliefs, desires and intentions as mental attitudes that deliberate human action. Rao & Georgeff [15] adopted this model and transformed it into in a formal theory and an execution model for BDI agents that serves as a basis for the implementation of several BDI agent platforms.

Examples of agent platforms that implement the BDI architecture include Jason [2], JACK [7], Jadex [14], and the 3APL Platform[3]. In particular, these four platforms are based on the Java language. However, even though the underlying language is a general purpose programming language, agents are implemented in these platforms in a new programming language – AgentSpeak(L) [16], JACK Agent Language, a Domain-specific Language (DSL) writen in XML, and 3APL [5], respectively. Source code written in these languages is either precompiled or processed at runtime by the agent platform. As a consequence, the adoption of this approach prevents developers from using advanced features of the Java language, such as reflection and annotations, and it makes it complicated to integrate the implementation of a multi-agent system with existing technologies. Both issues are essential in the context of the development of large scale enterprise applications. Due to these limitations of existing platforms, we have implemented a new BDI agent platform, namely BDI4JADE. Our implementation is a layer on top of an existing agent platform, JADE [1], which provides a robust infrastructure to implement agents, but not follow the BDI architecture. We built a BDI reasoning mechanism for JADE agents implemented directly in the Java language, thus addressing the aforementioned problems.

The remainder of this paper is organized as follows. We first provide an overview of the BDI4JADE in Section 2, and then detail its individual components in Section 3. Section 4 discusses relevant aspects of our BDI implementation on top of JADE, followed by Section 5, which describes related work. Finally, Section 6 presents final remarks.

2 BDI4JADE: an Overview

As stated in the introduction, our motivation for implementing a new BDI agent platform is that the languages provided by existing platforms, even though based on general purpose programming languages, limit integration with up-to-date available technologies, and also the use of advanced features of the underlying programming language. We faced problems of this nature while implementing different multi-agent systems [11–13], and also in our current research [10], which involves dynamic adaptations of BDI agent architectures. These problems are detailed in Section 4.

An agent framework that fulfils the requirement of not relying on a DSL is JADE [1]. JADE is not based on the BDI model, which is our target architecture, but implements a task-oriented model, in which agents have a set of behaviors. No cognitive abilities, such as a reasoning cycle, are provided for agents. However, JADE is a robust and mature infrastructure, and provides many features that are needed for implementing multi-agent systems, which include the yellow pages service and message exchange. In addition, it provides a behavior scheduler that can be used to control the execution of plans of BDI agents. So, instead of developing an agent platform from scratch, we implemented the BDI architecture as a layer on top of JADE. Agents implemented with BDI4JADE use only the constructions provided by the Java language, which makes it easy to integrate with existing applications and reusable software assets (frameworks, components, libraries). Next, we briefly introduce the main BDI4JADE components.

[3] http://www.cs.uu.nl/3apl/

BDI agent. A BDI agent represents an agent that follows the BDI architecture. It aggregates a reasoning cycle, responsible for driving agent behavior, strategies, and capabilities.

Capability. A BDI agent does not directly include a belief base and a plan library, but these are part of a capability. A capability [4] is a self-contained part of an agent, consisting of (i) a set of plans, (ii) a fragment of the knowledge base that is manipulated by these plans and (iii) a specification of the interface to the capability. Capabilities have been introduced into some multi-agent systems as a software engineering mechanism to support modularity and reusability, while still allowing meta-level reasoning.

Strategies. A BDI agent is associated with different strategies, which are points for customizing the reasoning cycle, and can have their default behavior modified by developers. They are related to the revision of beliefs, the generation and deliberation of goals and selection of plans.

Goal. Goals represent the motivational state of the system. It is an entity that represents a desire that the agent wants to achieve.

Intention. An intention captures the deliberative component of the system. An intention is a goal that the agent is committed to achieve, i.e. when an agent has an intention, it will select plans to try to achieve this intention, until the associated goal is achieved, no longer desired or considered unachievable.

Belief Base and Belief. Beliefs represent environment characteristics, which are updated accordingly after the perception of changes on it. Beliefs can be seen as the informative component of the system. A belief base is a set of beliefs, each of which has a name and a value.

Plan Library and Plan. BDI4JADE provides an infrastructure to implement reactive planning systems, in which plans are not generated but selected from an existing plan library. Plans contain a set of actions and are executed with the aim of achieving a specific goal.

Events. BDI4JADE provides means for creating observers (listeners) of beliefs and goals, in order to notify them when these concepts are updated, so they can update their state accordingly. Any component that registers itself as an observer is notified when beliefs are created, update or removed, and when goals changed their status.

These components are used in the reasoning cycle of our BDI agents, which is based on the BDI-interpreter algorithm presented in [15]. This cycle is implemented in six major steps:

1. *Revising beliefs.* This first step of the cycle consists of revising agent beliefs. In the default implementation, nothing is done at this step, but developers can specify a customized strategy for specific agents.
2. *Removing finished goals.* Before the cycle is executed, goals might have "finished," i.e. they may be achieved, no longer desired or considered unachievable. These are removed from the set of goals of the agent, and observers of these goals are notified about the event.
3. *Generating options.* In this step, the goals available to the agent are determined (its desires). It can generate new desired goals, determine that existing goals are no longer desired, or keep existing goals that are still desired.

4. *Removing dropped goals.* When a goal, or set of goals, is determined as no longer desired in the previous step, it is removed from the set of goals of the agent, and observers are notified about the occurrence of this event.
5. *Deliberating goals.* In this step, the current agent goals are partitioned into two subsets: (i) goals to be tried to be achieved (intentions); and (ii) goals to *not* be tried to be achieved. The last will remain as an agent desire, but the agent is not committed to achieve it at the moment.
6. *Updating goals status.* Based on the partition performed in previous step, the status of the goals are updated. Selected goals are updated to the status of trying to achieve, and unselected goals are updated to the status of waiting. When a goal has the status trying to achieve, the agent will select plans for achieving that goal.

3 Detailing BDI4JADE Components

The previous section provided an overview of the main components of our implementation of the BDI architecture, which was slightly modified, for instance, by the addition of capabilities. It also described the implemented reasoning cycle in a high-level way. In this section, we provide further details of our JADE extension, 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.

Most os the concepts presented in previous section, and their relationships, are depicted in Figure 1, which shows the class diagram of BDI4JADE. Due to space restrictions, it contains only the main components of our implementation, and it presents only methods from interfaces, and not from classes.

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. Therefore, as JADE agents, a BDI agent has its own thread of control, managed by 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 intention is created and attached to it. Intentions have a status associated with them, 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 architecture, an intention is a goal that an agent is committed to achieve. Our implementation does not make this distinction explicitly, but implicitly. Table 1 shows how concepts of the BDI architecture are related to the status of BDI4JADE

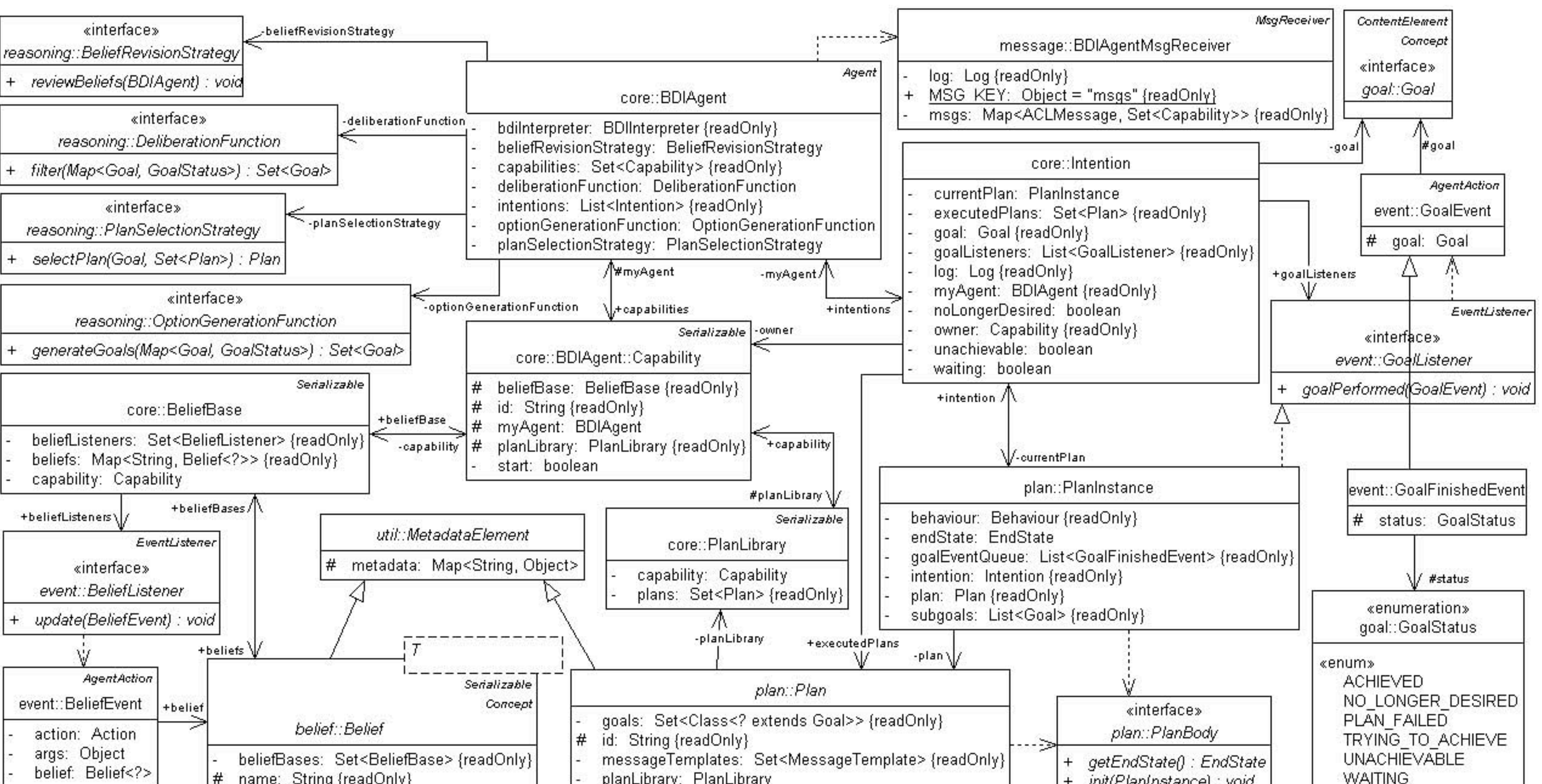

Fig. 1: Class diagram – BDI4JADE main classes and interfaces.

Status of the BDI4JADE Intention	BDI Architecture Concept
Waiting	Goal
Plan failed	Intention
Trying to achieve	Intention
Achieved	- (was an Intention)
No longer desired	- (was an Intention)
Unachievable	- (was an Intention)

Table 1: Intention Status x BDI Architecture Concept.

intentions. This approach was chosen to facilitate the implementation of the reasoning cycle. The last three status shown in Table 1 represent intentions/goals in a final state, and intentions with such status are removed from the agent in the next reasoning cycle.

As introduced before, beliefs and plans are not part of an agent (directly), as proposed in the BDI architecture, but part of capabilities. This concept is also implemented by JACK and Jadex agent platforms. As opposed to these platforms, 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. Shared belief bases are also possible.

A capability of our JADE extension is essentially composed of a belief base and plan library. The first is a collection of beliefs (see Section 3.3), and the latter a collection of plans (see Section 3.4). BDI4JADE does not provide means for explicitly defining capability interfaces, but they are exposed by documenting the capability. As a capability is associated with a set of plans, and these in turn are associated with the goals they can achieve, this set of goals indicates the goals that the capability can achieve. In addition, plans of a capability might require that other subgoals must be achieved when they are executed, so this set of goals indicates the goals that external components should achieve in order for the capability to be able to execute properly. These two set of goals can be seen as the provided and required interfaces of the capability, and should be part of the capability documentation.

All these components – capability, belief base and plan library – can be implemented either by *extension* or by *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 through 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 goals (which can be unknown at development time) can be added (desired) and achieved at runtime. This does not prevent the addition of goals at the agent initialization.

Reasoning Cycle. An essential part of a BDI agent platform is the reasoning cycle that it provides as part of agents. We previously presented how we implemented it in a high-level way. Next, we provide additional details. Listing 1.1 shows the source code of the reasoning cycle implemented in our platform.

Listing 1.1: 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();
      }
  }
}
```

The first step (line 2) invokes the belief revision function. It is performed by invoking the method void reviewBeliefs(BDIAgent) of an implementation of the BeliefRevisionStrategy interface. Next (lines 6-21), all finished intentions, i.e. intentions whose status is *achieved*, *no longer 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.

The method Set<Goal> generateGoals(Map<Goal, GoalStatus>) of an instance of the OptionGenerationFunction interface is then invoked (lines 23-24) to create new goals or to drop existing ones. 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 but not removed goals remain unchanged. The goalStatus is then updated (lines 44-47).

Next, it is time for the deliberation process, in which the agent selects the goals it will be committed to achieve. This is performed by invoking the method Set<Goal> filter(Map<Goal, GoalStatus>) of an instance of the deliberation function interface (line 48). It selects a set of goals that must be tried to achieve (intentions) from the set of goals. Selected goals and associated intentions 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 lines 51 and 53 correctly adjusts the new state of the intention.

This reasoning cycle is implemented as part of 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 this behavior to be continuously executed while there are no intentions and goals. In case a new intention is added to the agent, the reasoning cycle is resumed.

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>) of the plan selection strategy of the agent. After the plan selection, it will be instantiated and started.

Extension points. While describing the implemented reasoning cycle, we mentioned four strategies: BeliefRevisionStrategy, OptionGenerationFunction, DeliberationFunction and PlanSelectionStrategy, but we did not detail 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`: the `void reviewBeliefs()` method of the `BeliefBase` class of all capabilities is invoked;
- `DefaultOptionGenerationFunction`: it returns the current set of goals, i.e. it does not drop any of them and does not create any new goal;
- `DefaultDeliberationFunction`: it returns the whole set of goals, i.e. all goals will be set to a trying to achieve status; 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 [6].

3.2 Goals

A goal in BDI4JADE can be any Java object, with the condition that it must implement the `Goal` interface. Therefore, a class implementing this interface can be created and attributes can be added to it as inputs and outputs of the goal. We also provide a set of predefined 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 agent's 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 agent's 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. It is a subclass of the *`CompositeGoal`*.

`SequentialGoal`. This class represents a goal that aims at achieving all goals that compose it in a sequential way. It is a subclass of the *`CompositeGoal`*.

`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)` of an instance of the `BDIAgent`.

3.3 Beliefs

The `BeliefBase` class offers methods to manipulate beliefs, such as add, remove and update 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, as it is the case in Jadex. We have used Java generics to capture incorrect castings at compile time, so beliefs in the BDI4JADE are instances of subclasses of `Belief<T>`.

A belief has two 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 forgotten 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 to store belief values. The BeliefSet<T> defines methods to retrieve, store and iterate belief values, and has an implementation that stores values in memory – the TransientBeliefSet<T> class.

3.4 Plans

The representation of plans in the BDI4JADE is not associated with one but with a set of classes. One of the reasons is that our goal is to reuse JADE as much as possible in order to: (i) facilitate the learning process of developers already familiar with JADE; (ii) take advantage of the family of JADE behaviors; and (iii) 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, and their execution is controlled by the JADE scheduler.

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 has some information about it, which is: (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 (verification made at runtime). This method must be implemented, because it is an abstract method, and therefore the Plan class is also abstract.
- protected void initGoals() – this method must be overridden by subclasses to initiate the set of type 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 process.
- protected boolean matchesContext(Goal goal) – this method verifies a context to determine if the plan can achieve the goal according to the

Listing 1.2: Verifying if a plan can achieve a goal.

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

current situation of the environment. The default implementation returns always `true`.

Listing 1.2 presents the method that is executed to verify if a plan can achieve a given goal. If the goal is an instance of `MessageGoal`, i.e. it is the goal of processing a received message, it verifies if any of the message templates of the plan matches the received message. Otherwise, it checks if the goal has a type that can be achieved by the plan, and if so, it verifies if the context required by the plan matches the current context.

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, an 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). In addition, we also provide plans for achieving *`CompositeGoal`* goals.

`PlanInstance`. This class, as the name indicates, is an instance of a plan, which 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.

`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, due to Java limitations regarding multiple inheritance. So, our decision was to define an interface to be implemented by plan bodies, 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`. This shows that the platform detects that a goal was achieved when the selected plan finished with a

Listing 1.3: Dispatching and waiting for subgoals.

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

`SUCCESSFUL` state; 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.

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 1.3 shows an example of how it can 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 using the `receive(MessageTemplate)` method 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 that can process the message to perform it.

3.6 Events

Our platform implements the observer design pattern [6] in some points 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 of the `Belief` class, 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. An example of its use was presented in Section 3.4, in order to detect when a subgoal is achieved (or finished with another status).

4 Discussion

In this section, we discuss relevant aspects related to our JADE extension. These aspects are mainly associated with current limitations of BDI4JADE and development experiences with it.

Not implemented yet. There are improvements that we aim at developing for BDI4JADE, but they have not been implemented yet and will be future extensions of the platform. They are: (i) **Persistent beliefs** – currently, our platform only provides transient beliefs. We intend to incorporate the Hibernate[4] framework to our platform to facilitate the creation of beliefs that are persisted in databases; (ii) **Control of intention/goal owners** – we have created the `InternalGoal` interface 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. 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 internal goal. This helps creating encapsulated capabilities and improving reuse; and (iii) **Indexes for plan libraries** – every time that 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.

As we have not implemented (ii) yet, we also did not consider nested capabilities. The difference between adding two capabilities to an agent, or adding one capability to another, and the last to an agent is that when an internal goal is dispatched by the parent capability, it can be achieved by the plans that are part of it, or part of sub-capabilities. Without goal owners control, nested capabilities will present the same behavior of capabilities added to the same agent.

Debugging BDI4JADE agents. Most of existing BDI platforms provide tools to debug the implemented multi-agent systems and to inspect current state of agents. We have not developed any tools for supporting the development of agents. Nevertheless, as BDI4JADE agents are fully developed with Java, its debugger already provides information for debugging agents. The agent current state can be inspected with existing tools, typically attached to Java IDEs. In addition, tools provided by the JADE platform can also be adopted. They allow not only monitoring messages exchange, but also active plans, as they are implemented as JADE behaviors.

Testing BDI4JADE. In order to test our implementation, we have developed several example applications that test different parts of BDI4JADE. The tests included messages

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

exchange (ping application), different aspects of the reasoning cycle (trying different plans, dropping goals, and so on), and subgoals and composite goals. In addition, we have implemented the typical BDI application "Blocks world," which consists on moving blocks in an initial configuration to a target configuration. Moreover, BDI4JADE is been used in the context of our current research work [10]. It involves the development of agent-based software to assist users in routine tasks that users can customize based on a high-level language. This requires dynamic adaptation of agents architectures, and for that we adopt enterprise frameworks such as the Spring framework,[5] and therefore having agents implemented in "pure" Java is essential.

We have not run any stress test in BDI4JADE in order to test its scalability and performance, and compare these aspects with other existing platforms. We did not prioritize this kind of test because our main motivation with this work is to improve the development of multi-agent systems from a developer perspective, but, as thread control in BDI4JADE is performed by JADE, and this is the main issue related to the performance of multi-agent systems, we believe that systems implemented with BDI4JADE tend to have a performance similar to the ones implemented with JADE. However, further studies must be performed in this direction.

Relevance of the Integration with Existing Technologies. Our major concern while developing BDI4JADE was to provide an infrastructure that can be easily integrated with existing technologies. We identified this need during the development of multi-agent applications [11–13]. They involve the development of web-based systems that require the integration with: web application frameworks (help on managing web requests and creating dynamic web pages); Spring framework[5] (provides transaction management and dependency injection); software aspects [8] (a modularization technique for crosscutting concerns); and persistence frameworks (deal with database access and persisting entities). These technologies are commonly used for developing large scale enterprise applications and they are essential to increase software quality and reduce development costs, as these technologies have already been widely tested and provide ready-to-use infrastructures. In addition, software evolution is a reality, and agent technology must be able to be smoothly integrated to existing systems.

The main issue related to the integration with software frameworks is that, as opposed to libraries that are invoked by specific applications, they adopt the Hollywood principle: "Don't call us, we call you." This means that application-specific components are instantiated and invoked by the framework, and this usually requires components to implement interfaces. With existing agent platforms, this is not possible because components are not implemented with Java classes, and also the platform components are instantiated and manipulated by the agent platform, and its it usually hard to identify pointers to the platform component instances to make advances manipulations with them. In the case of software aspects, one of the most widely used implementations of it, AspectJ[6], requires exposing Java interfaces for the specification of join points, and again, without the implementation of Java classes, this is not possible.

[5] http://www.springsource.org/
[6] http://www.eclipse.org/aspectj/

5 Related Work

Different BDI agent platforms have already been proposed. Nevertheless almost all of them require the implementation of agents in a new programming language or a DSL, even though the implementation of the underlying agent platform is expressed in a general purpose programming language. This is the case of Jason [2], whose agents are implemented in an extension of the AgentSpeak language [16]; JACK [7], that has an specific language, the JACK Agent Language, which is precompiled for Java; and 3APL [5], an agent programming language with a platform implemented in Java.

The framework that has more similarities to BDI4JADE is Jadex [14], which uses JADE as a middleware. Our experience with the development of different applications using Jadex [11, 12] was also a motivation for developing our implementation. The main benefit of Jadex is that it provides the concepts of the BDI architecture for developers. In addition, it provides the capability concept, which allows for packaging a subset of beliefs, plans, and goals into an agent module and to reuse this module wherever needed. As a consequence, one can easily (un)plug capabilities to agents and reuse them.

However, Jadex defines agents through XML files, and this leads to drawbacks during the implementation. Programming an agent using XML prevents the use of features of the underlying programming language, and the integration with existing technologies becomes a challenge. Another disadvantage is that 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 only says 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.

6 Final Remarks

In this paper we presented BDI4JADE, an agent platform that implements the BDI architecture. As opposed to different BDI platforms that have been proposed, it does not introduce a new programming language nor rely on a DSL written in terms of XML files. Because agents are implemented with the constructions of the underlying programming language, Java, we bring two main benefits: (i) features of the Java language, such as annotations and reflection, can be exploited for the development of complex applications; and (ii) it facilitates the integration of existing technologies, e.g. frameworks and libraries, what is essential for the development of large scale enterprise applications, which involve multiple concerns such as persistence and transaction management. This also allows a smooth adoption of the agent technology. BDI4JADE is a BDI layer on top of JADE, and it leverages all the features provided by the framework and reuses it as much as possible. Other highlights of our JADE extension, besides providing BDI abstractions and reasoning cycle, include: (i) use of capabilities: agents aggregate a set of capabilities, which are a collection of beliefs and plans. This allows modularizing

particular behaviors of agents; (ii) 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 at compile time; and (iii) plan bodies as instances of JADE behaviors: in order to better exploit JADE features, in particular its behaviors hierarchy, plan bodies are subclasses of JADE behaviors. Our platform as well as examples of its use are available in [9]. BDI4JADE is being used in the context of our current research work [10].

References

1. Bellifemine, F.L., Caire, G., Greenwood, D.: Developing Multi-Agent Systems with JADE (Wiley Series in Agent Technology). John Wiley & Sons (2007)
2. Bordini, R.H., Wooldridge, M., Hübner, J.F.: Programming Multi-Agent Systems in AgentSpeak using Jason. John Wiley & Sons (2007)
3. Bratman, M.E.: Intention, Plans, and Practical Reason. Harvard University Press, Cambridge, MA (1987)
4. Busetta, P., Howden, N., Rönnquist, R., Hodgson, A.: Structuring BDI agents in functional clusters. In: ATAL ’99. pp. 277–289 (2000)
5. Dastani, M., van Riemsdijk, M.B., Dignum, F., Meyer, J.J.C.: A programming language for cognitive agents goal directed 3apl. In: Programming Multi-Agent Systems, LNCS, vol. 3067, pp. 111–130. Springer Berlin / Heidelberg (2004)
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-oriented Software. Addison-Wesley (1995)
7. Howden, N., Rnnquista, R., Hodgson, A., Lucas, A.: Jack intelligent agents™: Summary of an agent infrastructure. In: The Fifth International Conference on Autonomous Agents. Montreal, Canada (2001)
8. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-Oriented Programming. In: ECOOP 1997. vol. 1241, pp. 220–242. Springer-Verlag, Berlin, Heidelberg, and New York (June 1997)
9. Nunes, I.: A bdi extension for jade (2010), http://www.inf.puc-rio.br/˜ionunes/bdi4jade/
10. Nunes, I., Barbosa, S., Lucena, C.: Increasing users’ trust on personal assistance software using a domain-neutral high-level user model. In: ISoLA 2010, LNCS, vol. 6415, pp. 473–487. Springer (2010)
11. Nunes, I., Cirilo, E., Lucena, C.: Developing a family of software agents with fine-grained variability: an exploratory study. In: SEAS 2009. pp. 71–82 (2009)
12. Nunes, I., Kulesza, U., Nunes, C., Cirilo, E., Lucena, C.: Extending web-based applications to incorporate autonomous behavior. In: Proc. of the 14th Brazilian Symposium on Multimedia and the Web (WebMedia’08). pp. 115–122 (2008)
13. Nunes, I., Nunes, C., Kulesza, U., Lucena, C.: Agent-oriented software engineering ix. chap. Developing and Evolving a Multi-agent System Product Line: An Exploratory Study, pp. 228–242. Springer-Verlag, Berlin, Heidelberg (2009)
14. Pokahr, A., Braubach, L.: Jadex user guide. Tech. Rep. 0.96, University of Hamburg, Hamburg, Alemanha (2007)
15. Rao, A.S., Georgeff, M.P.: BDI-agents: from theory to practice. In: Proceedings of the First Intl. Conference on Multiagent Systems. San Francisco (1995)
16. Rao, A.S.: Agentspeak(l): Bdi agents speak out in a logical computable language. In: MAAMAW ’96. pp. 42–55. Springer-Verlag (1996)
17. Zambonelli, F., Jennings, N.R., Omicini, A., Wooldridge, M.: Agent-oriented software engineering for internet applications. In: Coordination of Internet Agents, pp. 326–346. Springer Verlag (2001)