Norm-based Behavior Modification in Reflex Agents
An Implementation in JAMDER 2.0

Francisco I. S. Cruz, Robert M. Rocha Jr, Emmanuel S. S. Freire and Mariela I. Cortés
GESSI - Grupo de Engenharia de Software e Sistemas Inteligentes, Computer Science Department, Universidade Estadual do Ceará (UECE), Fortaleza, Ceará, Brazil

Keywords: Normative Simple Reflex Agent, Norms, Framework, Normative Multi-agent Systems.

Abstract: The agent-oriented development is becoming more frequent in the industry and academy. Currently, more works are turning to the growth of this area. Many frameworks that support the development of Normative Multi-agent Systems. However, few works deal the impact of norms on individual behavior of the agent. Like many others, JAMDER 2.0 framework follows this aspect. This paper discusses the modification of the behavior of simple reactive agent based on impact caused by norms on the JAMDER 2.0 platform. This work has been collaborating for the extension of this framework, re-establishing the dynamism, which was in its first version, and giving it support for changing the behavior of simple reactive agent. In addition, new features have been included in the framework. Among them, an agent that is able to monitor the actions of a set of agents, evaluating them according to the norms and applying appropriate sanctions to these agents, if available. For illustrate extension, the Vacuum cleaner world was implement using the extended JAMDER 2.0.

1 INTRODUCTION

The agent-oriented development is becoming more frequent in the industry and academy (Silva and Castro, 2002). In order to cope with the heterogeneity in these systems, governance mechanisms are defined through a set of norms that must be attended by the entities in the system. Norms promote modifications in the agent decision processes and they can influence in the rational behavior and performance of the agent. In this sense, a thorough understanding of the impact of norms on the level of individual agents is critical in order to deal with the dynamic aspects of normative multi-agent systems (NMAS).

Similarly to JAMDER (Lopes et al., 2011), so many frameworks assist agent-oriented development. However, few research efforts have addressed the impact of norms on individual agent behaviour (Campos, Freire and Cortés, 2012). In the JAMDER 2.0 framework the deontic concepts of norms are contemplated, however, only the case of goal-oriented agents with plans is considered. Thus, an adequate mechanism to deal with norms in the other kinds of intelligent agents is missing. So, this paper is a contribution in order to provide support to the implementation of the static aspects of norms in simple reflex agents.

The paper is structured as follows. Section 2 presents the related work. The concepts related to normative MAS and the JAMDER 2.0 framework are detailed in Section 3. Section 4 describes the proposed framework´s extensions. A case study is showed in Section 5 and, finally, conclusions and future work are presented in Section 6.

2 RELATED WORK

A set of frameworks and platforms has been developed to support the development of MAS´s. In general, these mechanisms are associated with a programming language for composing entities and provide an environment for their execution. Following are some of the most used frameworks for implementing MAS.

JACK (Jack, 2013) is a framework in Java for development of MAS. It provides high performance, and an easy way to be extended to support BDI agents and specifics requirements of applications. The language used by JACK is built from Java language and can be used in the development of BDI agents and their behavior. However, JACK does not support the modeling of normative concepts neither the simple reflex agent and its IDE is not freeware.

The development of JAM (Huber, 2013) was based on a series of theories and frameworks for

Cruz F., Rocha Jr R., Freire E. and Cortés M..
Norm-based Behavior Modification in Reflex Agents - An Implementation in JAMDER 2.0.
DOI: 10.5220/0004899004970504
In *Proceedings of the 16th International Conference on Enterprise Information Systems* (ICEIS-2014), pages 497-504
ISBN: 978-989-758-027-7

agents based on BDI agents. It allows the implementation of agents that have plans and goals. However, JAM does not have development tool, does not allow implementation of agent role, organization and the normative concepts.

JADE (Jade, 2013) is framework implemented in Java language that simplifies the development of MAS through middleware that complies with the FIPA specifications and with a set of graphical tools that support debugging. This framework allows the implementation of typical elements that compose MAS. However, it does not have support for normative concepts and simple reflex agent.

How JADE did not support the implementation of normative concepts, Rocha Júnior, Freire and Cortés (2013) proposed its extension, called JAMDER 2.0. This extension was based on modeling language NorMAS-ML (Freire et al., 2012). Thus, JAMDER 2.0 besides supporting the modeling of the typical elements of MAS along with normative concepts. However, JAMDER 2.0 do not support the modeling of the agent architectures defined by Russell and Norving (2003) considering the norms for execute their actions.

Normative Jason (Santos Neto, 2012) is an extension of Jason framework based on AgentSpeak Normative language). This framework provides the implementation of normative agents based in BDI agents which are able to understand, to follow or to violate the norms contained in the environment. However, the language is defined on first-order logic, the focus of norms is the behavior of agents and not in other entities that compose normative MAS, and does not have support to exchange roles between agents neither the simple reflex agent.

Based on the analysis presented and considering the need of a framework that enables the implementation of normative simple reflex agent together with the entities that compose a MAS, JAMDER 2.0 is highlighted because (i) It has a platform on JAVA language, (ii) It has support to distributed system, (iii) It complies with the FIPA patterns, (iv) It supports the MAS typical entities and the normative concepts, and (v) It has graphical interface and plugins for IDEs.

3 BACKGROUND

3.1 Normative Simple Reflex Agent

The agent architecture purely reflex agents should be able to quickly respond to changes in the environment by means of its condition-actions rules. With this, the agent perceives information about the state of the environment through sensors and based on rules in the form “if condition then action”, it selects the most adequate action for the current perception. The agent performs the selected action upon the environment through actuators.

This kind of agent may be inserted into an environment that has a specified set of norms that restrict their actions. As defined by Figueiredo and Silva (2011), the norms are intended to restrict the behavior of agents applying sanctions when they are violated or fulfilled.

Therefore, the norms of an environment should not be able to avoid the execution of certain action, but rather to penalize or reward an agent if the action taken by it is prohibited or obligated. Therefore, if the set of norms defined in the environment is not considered in the condition-action rules, an agent can be penalized if it performs a prohibited action.

In order to avoid the violation of the simple reflex agent architecture, Campos, Freire and Cortés (2012) propose to consider the information about the set of norms as an extension of the condition-action rules. It involves the definition of three different groups of condition-action rules. Each group is associated with one deontic concept and considers the sanctions linked to each norm, that is:

- Obligation Rules Group: specifies the rules related with the actions that must be performed by the agents. If an event of environment matches with a rule in this group, it must necessarily be performed by the agent;
- Prohibition Rules Group: specifies the rules that are related with the actions that cannot be performed by the agent. If an event of environment matches with a rule in this group, the rule will not be executed by the agent;
- Permission Rules Group: specifies the rules related with the actions that can be executed. If an event of environment matches a rule set out of this group, it may or may not be executed by the agent.

In the architecture were added two new groups in the agent´s action selection mechanism, corresponding to the representation of the information about its obligations and prohibitions. This approach considers that if an action is obligated, then the agent must perform that action only if it is not prohibited. If an action is prohibited, then the agent must perform another action, different from the prohibited action, which is permitted and rational. If there is not an action that is obligated and prohibited, then the agent must perform a permitted action which is rational, as would do a well-designed simple reflex agent in an environment without norms.

3.2 JAMDER 2.0 Framework

JAMDER 2.0 (Rocha Júnior, Freire and Cortés, 2013) is an extension of JAMDER (JADE to MAS-ML 2.0 Development Resource) (Lopes et al., 2012) that incorporates the resources offered at modeling level by the NorMAS-ML language, including, specifically, the norms and their properties.

The extension process was characterized by the mapping between concepts of modeling and implementation in order to identify which elements of the conceptual metamodel relating to NorMAS-ML entities were present in JAMDER. Therefore, the following set of classes was associated with JAMDER for representing the normative concepts:

- Norm: For representing the norm entity was created the class jamder.norms.Norm which defines the following properties: (i) the identifier, (ii) the norm type, (iii) the restricted entity, (iv) the context, (v) the action, (vi) and the list of activation constraints.
- NormResource: A norm is set to restrict the behavior of a given resource (Freire et al., 2012). The jamder.norms.NormResource class was created to represent any case of norm resource if it is: (i) a structural feature; (ii) a behavioral feature; (iii) an entity; (iv) a relationship or (v) a message;
- NormAction: The actions linked to the norms were represented by the main class jamder.norms. NormAction and two sub-classes: jamder.norms. AtomicAction and jamder.norms.CompositeAction, which represent the operation system.
- NormConstraint: The jamder.norms. NormConstraint class was created to represent the activation constraints with their respective sub-classes that include each type of constraint: jamder.norms.Before, jamder.norms.After, jamder. norms.Between and jamder.norms.IfConditional. It is directly linked to class jamder.norms.Date responsible for setting a date.

Some JAMDER classes were modified to include the properties defined in the language NorMAS-ML. We describe these changes as follows:

- The list of norms contextNorms was added in the jamder.Environment and jamder.Organization classes. It represents norms whose environment or organization is defined as context.
- The list of norms restrictNorms was added in the jamder.Environment, jamder.Organization, jamder. agents.GenericAgent and jamder.roles.AgentRole classes. It represents norms whose instances of class is a restricted entity.
- Finally, the following classes representing instances of duty, right, and axiom, respectively, jamder.behavioural.Duty, jamder.behavioural.Right and jamder.structural.Axiom were removed along with the jamder.behavioural.DutyRightProperty class. It is necessary because these concepts are considered semantically equivalent to the deontic concepts of permission and obligation in norms and are removed in NorMAS-ML.

With these changes in some JAMDER classes and the creation of new classes (defined in the previous subsection), the JAMDER 2.0 framework allows the modeling properties and relationships of entities within a normative multi-agent system.

4 EXTENSION OF JAMDER 2.0

In order to restore the dynamics of the tool JAMDER 2.0 and, establish when a norm would be active, it has been required some changes in the code and the addition of some methods, classes and relationships between classes.

A norm can be active depending on the execution of a NormAction (Rocha Jr., Freire and Cortés, 2013) then the Action class (which is a representation of AgentAction) began to relate with the NormAction class in order to map when a NormAction linked to an Action is executed. Whenever an action is executed (as an event), Action sends a signal to NormAction.

The Property class (Booch et al., 2000) received the concept of Java Generics (Oracle, 2013) to make possible comparisons between properties in the IfCondition class, that inherits from NormConstraint, whose function is to abstract the activation conditions of norms (Rocha Júnior, Freire and Cortés, 2013). For instantiating a Property, it should use the following structure:

```
Property<T> propertyName=new
Property<T>();
```

Where T must inherit from Comparable class (Oracle, 2013).

The abstract method isTrue() was added in NormConstraint class. Its function is to enable the statement of when a constraint is true and thus enable verification of time when a norm is active. This method was implemented in the classes: IfCondition, Before, After and Between that inherit from NormConstraint (Rocha Júnior, Freire and Cortés, 2013).

In the Norm Class methods were added:

- setContext (Object) and setRestrict (Object) - their function is modify the context and the entity restricted of a norm, generically, without having to know which instance of them;

- getContext () and getRestrict () - it returns an Object that contains the context and the restricted entity respectively;
- isActive() - which specifies when the norm is active or not;
- apply() and disapply() - which switching on/off a norm from their context and restricted entity (apply and disapply a norm). These methods are important in sanctions application since a sanction in JAMDER 2.0 is a norm (Rocha Júnior, Freire and Cortés, 2013);
- isApply() - it indicates if a norm has been applied.

Through the methods setContext(object) and getContext(Object), the norm knows the restricted actions that are in its context. On the other hand, the methods setRestrict(Object) and getRestrict(Object) indentify the entities restricted by a norm. In addition, the methods apply() and disapply() briefs what norms are active and inactive in the environment. It is fundamental for the agent behavior modification because the agent need to know what norms are restricted it. For instance, when a new norm is added in an organization and this norm refers to a specified agent, this norm is added in the organization and in the agent. With this, the context and the envolved entities know the norms that restricts them.

Consequently, where a new norm are added in environment, these methods ensure that the norm list of the restrict entities will be update.

The NormResourceProperty class that inherits from NormResource was created in order to supply a resource of norm linked to a property. Differently from NormResource, NormResourceProperty carries the concept of Generics in Java such as Property class.

In the AgentRole and ReflexAgentRole classes, the method inicialize(), before disabled, has been revitalized. This was only possible because of inclusion of deontic concept in Action class, since an agent role consists of a set of rights (which become permission) and duties (which become obligation) and the deontic concepts of a norm replace them (Freire et al., 2012).

The Action class receives a NormType as an attribute in order to abstract the deontic concept in a condition-action rule and allowing the revitalization of AgentRole and, changing the reflex agent behavior.

Finally the ReflexAgent class receives methods containsNorm(Action, NormType) and containsNormDiferent(Action, NormType) that are responsible to check if there is any active norm (with the concept of deontic NormType) whose action is linked to the same or different Action, respectively. The Figure 1 shows the Reflex Agent Class.

```
package jamder.agents;

import jamder.Environment;
import jamder.behavioural.Action;
import jamder.behavioural.Sensor;
import jamder.norms.Norm;
import jamder.norms.NormType;
import jamder.roles.AgentRole;

import java.util.Hashtable;

public abstract class ReflexAgent extends GenericAgent {
    private static final long serialVersionUID = 1L;
    Sensor perception = new Sensor(this, 1000L);
    protected Hashtable<String, Action> perceives =
                        new Hashtable<String, Action>();
    protected ReflexAgent(String name,
            Environment environment, AgentRole agentRole) {
        super(name, environment, agentRole);
    }
    protected Action getPerceive(String perceive) {
        return perceives.get(perceive);
    }
    protected void addPerceive(String perceive, Action action) {
        perceives.put(perceive, action);
    }
    protected Action removePerceive(String perceive) {
        return perceives.remove(perceive);
    }
    protected void removeAllPerceives() {
        perceives.clear();
    }
    protected Hashtable<String, Action> getAllPerceives() {
        return perceives;
    }
    protected void setup() {
        super.setup();
    }
    protected void takeDown() {
    }
    public void percept(Object perception) {
        Action action=null;
        if (perception instanceof String &&
                    perceives.containsKey((String)perception)){
            action = perceives.get(perception);
            addBehaviour(action);
        }
    }
    public Norm containsNorm(Action ac, NormType nt){
        for (Norm nor: getAllRestrictNorms().values()){
            Action a=nor.getAction().getNormResource().getAction();
            boolean valida=a!=null &&
                    a.getName().equalsIgnoreCase(ac.getName()) &&
                    nor.getNormType()==nt &&
                    nor.isActive();
            if (valida){
                return nor;
            }
        }
        return null;
    }
    public Norm containsNormDiferent(Action ac, NormType nt){
        for (Norm nor: getAllRestrictNorms().values()){
            Action a=nor.getAction().getNormResource().getAction();
            boolean valida=a!=null &&
                    !a.getName().equalsIgnoreCase(ac.getName()) &&
                    nor.getNormType()==nt &&
                    nor.isActive();
            if (valida){
                return nor;
            }
        }
        return null;
    }
}
```

Figure 1: Reflex Agent Class.

4.1 Verification of Norms

The solution to behavior modification comes through the norm verification, present in ReflexAgent, before the execution of each action. This check was made on the action() method, from Action class. Before the execution of each action it makes the verification shown in pseudo code below:

```
if (There is an active obligation norm
related to the action that will be
executed){
   Executes the action independently
   of its preconditions.
}else{
   if ((there is no other active
   obligation norm that is not related to
   the action that will be executed) and
   (there is no an active prohibiting
   norm related to the action that will
   be executed)){
       Executes the action depending
       on preconditions.
   }
}
```

Thus, the agent always executes an obligation action if the obligation norm is active. Otherwise, it checks if other action, other than that it wants execute, is required by a norm. If so, it checks if the action is prohibited. Otherwise, it performs the action if their preconditions are met. If there is other action obligated by an active norm, it will be executed when the same verification above is done when its method action() is executed.

4.2 Monitoring Agent

In order to evaluate the simple reflex agent behavior modification, it is necessary somehow to monitor its behavior. Thus, we created a Monitor abstract class which inherits from GenericAgents and owning the method percept (Object, Object). This method is called by tied agent to the monitor when it (tied agent) executes an action. Thus it becomes possible to monitor and modify the behavior of simple reflex agent.

The main function of the monitoring agent is (i) to evaluate if the tied agent fulfillment or violation of a norm and (ii) to apply related sanctions when necessary.

The monitoring in a normative system requires an abstraction for the states of a norm. Then the literature indicates the use of an Augmented Transition Networks (ATNs) (Modgil et al., 2009). An ATN is a graph representing the three states a norm can take. They are INACTIVE, ACTIVE, COMPLIANCE-OR-INFRINGEMENT.

In order to incorporate this concept into JAMDER were created ATN classes and ATNState representing Augmented Transition Networks and their states respectively. The second is an enumeration of type String representing the three states mentioned above.

The Monitor class attributes are a set of agents (to be monitored) and the ATN which are generated by norms that restrict these agents whenever an agent is added. The behavior of the monitoring agent is up to the user. This is important because the literature often differs a lot about how and when sanctions are to be applied to the agent to be monitored (Piunti et al., 2010). The Monitor class (Figure 2) has the following methods:

- getAgents() - that returns the agents being monitored by the monitor;
- addAgent(String, GenericAgent) - which is used to add an agent to be monitored;
- addATN(Norm) - add a ATN by means of a norm. This method is used in apply() method of the Norm class. When a norm happens to restrict an agent, a new ATN is created for monitoring this agent;
- removeATN(String) – remove an ATN through its key. This method is used in disapply() method of the Norm class. When a norm fails to restrict an agent, its ATN is removed;
- getAllAtns() - that returns a Hashtable containing the ATNs linked to norms that restrict the agents being monitored;
- punish(Norm) - applying punishment (if it exists) linked to the norm that receives as the parameter;
- reward(Norm) - applying reward (if it exists) linked to the norm that receives as the parameter.
- percept(Object, Object) - It is an abstract method and is a implementation of GoF Observer pattern (Vlissides et al., 1995). When an action is executed, the agent sends a signal to its monitoring agent through this method.

5 CASE STUDY

This section shows the use of the JAMDER extension for implement the normative vacuum cleaner world. This case study was used by Campos, Freire and Cortés (2012) to present their approach.

Considering the normative vacuum cleaner world with only two rooms, where each room can be clean or dirty, in our experiments the perceived information of the environment were represented by the environmental states. In addition, this world has norms that restrict the agent's behavior for execute their activities, and is composed by:

Place - an object that is a graph with two vertices. Those vertices denote the room: roomA e roomB;

- CleanserOrg - an organization that has the agent roles: management e cleaner. The first role is linked to manager agent and the other is linked to VacuumCleaner;
- ManagerAgent - an agent of monitor type that use the management paper in the CleanserOrg organization. Its function is to monitor the Vacuum Cleaner agent and to modify its behavior as the

environmental norms. Its actions are ActionMonitorCyclic and ActionMonitorReflex;

- VacuumCleaner - a normative simple reflex agent that implements the cleaner paper in CleanserOrg organization. Its actions are Right, Left, Suck and NextAction.

The behavior of the ManagerAgent (Figure 3) is denoted for the actions (i) ActionMonitorCyclic, that is responsible for applying sanction if the agent has not executed a prohibited action or an obligated action, and (ii) ActionMonitorReflex, that is responsible for applying sanction if the agent has executed a prohibited action or an obligated action.

The Vacuum cleaner (Figure 4) has the actions Right, Left, Suck e NextAction. The NextAction is responsible for perceiving the environment and controlling the preconditions of the others actions:

- Right - its precondition is true if the vacuum cleaner is in RoomA and it is clean;
- Left - its precondition is true if the vacuum cleaner is in RoomB and it is clean;
- Suck - its precondition is true if the RoomA or RoomB is suck.
- NextAction - it is the agent perception core. This action constantly checks if the room is dirty and which room is the agent.

The world has the following norms:

- N1: the vacuum cleaner is required to suck the roomA from 4:00 to 6:00 a.m.;
- N2: the vacuum cleaner cannot suck the roomA from 1:00 to 3:00 a.m.;
- N3: if vacuum cleaner fulfills the N1 norm, it wins 3 points;
- N4: if vacuum cleaner fulfills the N2 norm, it wins 2 points.

In the beginning of each experiment the normative vacuum cleaner does not know the world configuration in terms of dirt. We considered that when the world is without the presence of norms, the measure of performance evaluation offers the reward of one point per each square clean (+1) and penalizes with the loss of one point per each movement (-1). In the case of the presence of norms in the world, the measure must be adapted in order to consider the rewards (+points) and the penalties (-points), which are consequences of the agent accepting or rejecting some norm.

```
import jamder.Environment;
import jamder.Organization;
import jamder.agents.GenericAgent;
import jamder.behavioural.Action;
import jamder.norms.Norm;
import jamder.roles.AgentRole;

import java.util.Hashtable;

public abstract class Monitor extends GenericAgent {
    private static final long serialVersionUID = 1L;
    private Hashtable<String, ATN> atns=new Hashtable<String, ATN>();
    private Hashtable<String, GenericAgent> agents=new Hashtable<String, GenericAgent>();


    public Monitor(String name, Environment environment, Organization owner) {
        super(name, environment, null);
        if (owner!=null && owner.getName()!=null)
            addOrganization(owner.getName(), owner);
    }

    public Hashtable<String, GenericAgent> getAgents() {
        return agents;
    }

    public void addAgent(String key, GenericAgent agent) {
        this.agents.put(key, agent);
        for (Norm nor: agent.getAllRestrictNorms().values()){
            ATN atn=new ATN(nor, nor.getName());
            atns.put(atn.getName(), atn);
        }
    }

    public void addATN(Norm nor){
        if (!atns.containsKey(nor.getName())){
            ATN atn=new ATN(nor, nor.getName());
            atns.put(atn.getName(), atn);
        }
    }

    public void removeATN(String key){
        atns.remove(key);
    }

    public Hashtable<String, ATN> getAllAtns() {
        return atns;
    }

    public void punish(Norm norm){
        Hashtable<String, Norm> punishments=norm.getSactionPunishment();
        for (Norm punishment: punishments.values()){
            if (punishment.isApply())
                punishment.disapply();
            Object context= norm.getContext();
            Object restrict=norm.getRestrict();
            punishment.setContext(context);
            punishment.setRestrict(restrict);
            punishment.apply();
        }
    }

    public void reward(Norm norm){
        Hashtable<String, Norm> rewards=norm.getSactionReward();
        for (Norm reward: rewards.values()){
            if (reward.isApply())
                reward.disapply();
            Object context= norm.getContext();
            Object restrict=norm.getRestrict();
            reward.setContext(context);
            reward.setRestrict(restrict);
            reward.apply();
        }
    }


    public abstract void percept(Object perception1, Object perception2);


}
```

Figure 2: Monitoring Agent Class.

```
import jamder.Organization;
import jamder.agents.ReflexAgent;
import jamder.behavioural.Action;
import jamder.manager.Monitor;
import jamder.roles.AgentRole;

public class ManagerAgent extends Monitor {


    private static final long serialVersionUID = 5670776386732896828L;

    public ManagerAgent(String name, Environment environment,
                        Organization owner) {
        super(name, environment, owner);
        Action ac=new ActionMonitorCyclic("ActionMonitorCyclic");
        addAction("ActionMonitorCyclic",ac);
        AgentRole ar=new AgentRole("management", owner, this);
        ar.addAction("ActionMonitorCyclic",ac);
        addAgentRole("management", ar);
        ar.initialize();
    }

    @Override
    public void percept(Object perception1, Object perception2) {
        if (perception1 instanceof ReflexAgent &&
                                perception2 instanceof Action){
            addBehaviour(new ActionMonitorReflex("ActionMonitor",
                (ReflexAgent)perception1, (Action)perception2));
        }
    }
}
```

Figure 3: ManagerAgent.

```
import jamder.Environment;
import jamder.agents.ReflexAgent;
import jamder.behavioural.Action;
import jamder.behavioural.Condition;
import jamder.manager.Monitor;
import jamder.norms.NormType;
import jamder.roles.AgentRole;

public class VacuumCleaner extends ReflexAgent{
    private static final long serialVersionUID = 1L;

    private Condition cleft=new Condition("cleft", null, false);
    private Condition cright=new Condition("cright", null, false);
    private Condition csuck=new Condition("csuck", null, false);
    private int points=0;

    protected VacuumCleaner(String name, Environment environment, AgentRole agentRole, Monitor monitor) {
        super(name, environment, agentRole);
        Action letleft=new Left("LetLeft");
        letleft.setCyclic(true);
        letleft.addPreCondition(cleft.getName(), cleft);
        letleft.setNormType(NormType.PERMISSION);
        Action letrigth=new Right("LetRight");
        letrigth.setCyclic(true);
        letrigth.addPreCondition(cright.getName(), cright);
        letrigth.setNormType(NormType.PERMISSION);
        Action letsuck=new Suck("LetSuck");
        letsuck.setCyclic(true);
        letsuck.addPreCondition(csuck.getName(), csuck);
        letsuck.setNormType(NormType.OBLIGATION);


        Action setup=new NextAction("setup", cleft, cright, csuck);
        setup.setNormType(NormType.OBLIGATION);
        setup.setCyclic(true);

        setMonitor(monitor);
        addAction(letleft.getName(), letleft);
        addAction(letrigth.getName(), letrigth);
        addAction(letsuck.getName(), letsuck);
        addAction("setup", setup);
    }

    public void setPoints(int points) {
        this.points = points;
    }


    public int getPoints() {
        return points;
    }

    @Override
    public void addAgentRole(String name, AgentRole role) {
        super.addAgentRole(name, role);
        role.initialize();
    }


}
```

Figure 4: VacuumCleaner.

5.1 Experiments

Firstly, the vacuum cleaner is in Normative Vacuum Cleaner World that only the norm N1 is active. Table 1 shows the execution of the agent.

Table 1: Running with just the norm N1.

State			Action	Score
Where is it?	State roomA	State roomB		
roomA	Dirty	dirty	suck	1
roomA	Clean	dirty	right	0
roomB	Clean	clean	suck	1
roomA	Clean	clean	suck	4
roomA	Clean	clean	right	3
roomB	Clean	clean	suck	2
roomA	Clean	clean	right	1

The rows one to three describe the behavior of the agent in a period in which the norm had not been activated and the agent was governed by the rules in the Permission Group. The row four of the table (shaded) illustrates the behavior of the agent when the norm N1 was activated. It is noticed that the agent was rewarded with three points per action during the period, according with the sanctions associated with the norm of obligation (N3). In rows five to seven, the norm was expired and the agent behavior was again governed by the Permission rules.

After, the vacuum cleaner is in Normative Vacuum Cleaner World that the norms N1 and N2 are active. Table 2 shows the execution of the agent.

Table 2: Running with norms N1 and N2.

State			Action	Score
Where is it?	State roomA	State roomB		
roomA	dirty	dirty	No_op	2
roomA	dirty	dirty	suck	3
roomA	clean	dirty	right	2
roomB	clean	clean	suck	3
roomB	clean	clean	suck	6
roomB	clean	clean	left	8
roomA	clean	clean	right	7

The rows one and six of the table illustrate the behavior of the agent when the norm N2 was activated and N1 was deactivated. It is noticed that the agent was rewarded with two points per action during the period, according with the sanctions associated with the norm of prohibition (N4). The rows two to four describe the behavior of the agent in a period in which the norms N1 and N2 had not

been activated and the agent was governed by the rules in the Permission Group.

The row five of the table illustrates the behavior of the agent when the norm N1 was activated and N2 was deactivated. It is noticed that the agent was rewarded with three points per action during the period, according with the sanctions associated with the norm of obligation (N3). In row seven, the norms N1 and N2 was expired and the agent behavior was again governed by the Permission rules.

6 CONCLUSION AND FUTURE WORK

The influence of the norm concepts related to the reflex agent architectures is essential in order to improve the performance of the agents executing in an environment governed by norms. In this context, this paper presented the extension of JAMDER 2.0 framework through a mapping between the characteristics of the approach proposes by Campos, Freire and Cortés (2012) and JAMDER 2.0. In addition, the monitoring agent was proposed in order to support the monitoring of the agents. Finally, a example based on a Normative Vacuum Clear World has been used to illustrate the use of the extension of JAMDER 2.0 framework, demonstrating its applicability and adequacy for developing normative simple reflex agent in NMAS.

As future work, it is relevant to consider (i) the automatic code generation from the normative simple reflex agent, based on extension of JAMDER 2.0 framework proposed in this work and (ii) the new extension of JAMDER 2.0 is required for that others agent architecture proposed by Russell and Norvig (2003) can understand the environmental norms.

REFERENCES

Booch, G., Jacobson, I., and Rumbaugh, J. (2000). Omg unified modeling language specification. Object Management Group ed: Object Management Group, page 1034.

Campos, G. A., Freire, E. S., and Cortés, M. I. (2012). Norm-based behavior modification in reflex agents. In: *International Conference on Artificial Intelligence (ICAI)*.

Figueiredo, K. and da Silva, V. T. (2011). Norm-ml: A modeling language to model norms. *ICAART* (2), edited by J. Filipe and ALN Fred, pages 232–237.

Freire, E. S. S., Cortés, M. I., Gonçalves, E. J. T., and Lopes, Y. S. (2012). A Modeling Language for Normative Multi-Agent Systems. *13th International Workshop on Agent-Oriented Software Engineering (AOSE@AAMAS)*, Valencia (Spain).

Huber, M. J.: JAM Agent. Available at: http:// www.marcush.net/IRS (2013).

JACK Agent Language. Available at: http:// www.agentsoftware.com.au/products/jack/ (2013).

Java Agent Development Framework. Available at: http://jade.tilab.com/ (2013).

Lopes, Y. S., Gonçalves, E. J. T., Cortés, M. I., and Freire, E. S. S. Extending jade framework to support different internal architectures of agents. *9th European Workshop on Multi-agent Systems (EUMAS2011)*, Maastricht, Holland.

Modgil, S., Faci, N., Meneguzzi, F., Oren, N., Miles, S.and Luck, M. (2009). A framework for monitoring agent-based normative systems. In *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems*-Volume 1, pages 153–160. International Foundation for Autonomous Agents and Multiagent Systems.

ORACLE, editor (2013). Generic types. Access date: 18 Oct. 2013.

Piunti, M., Ricci, A., Boissier, O., H¨ ubner, J. F., et al. (2010). Programming open systems with agents, environments and organizations. In: WOA 2010 - 11 Workshop nazionale'Dagli Oggetti agli Agenti'.

Rocha Jr., R. M., Freire, E. S. S., and Cortés, M. I. (2013). Estendendo o Framework JAMDER para Suporte à Implementação de Sistemas Multi-Agente Normativos. In: *IX Simpósio Brasileiro de Sistemas de Informação (SBSI)*, 2013, João Pessoa, Brasil. Anais do IX Simpósio Brasileiro de Sistemas de Informação (SBSI).

Russell, S. J. and Norvig, P. (2003). Artificial intelligence: A modern approach; [the intelligent agent book]. 2. ed. Prentice Hall series in artificial intelligence. Prentice Hall, Upper Saddle River, NJ.

Santos Neto, B. F.: Uma abordagem deôntica para o desenvolvimento de agentes normativos autônomos. Tese de doutorado. Rio de Janeiro: PUC, Departamento de Informática (2012), Brasil.

Silva, C. T. L. L., Castro, J. F. B. (2002). Modeling Organizational Architectural Styles in UML: The Tropos Case. In: *WER02 - V Workshop on Requirements Engineering*, 2002, Valencia, Spain.

Vlissides, J., Helm, R., Johnson, R., and Gamma, E.: Design patterns: Elements of reusable object-oriented software. Reading: Addison-Wesley (1995).