PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 30/05

Governing Agent Interaction in Open Multi-Agent Systems

Rodrigo de Barros Paes
Carlos José Pereira de Lucena
Paulo S. C. Alencar

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22453-900
RIO DE JANEIRO - BRASIL

Monografias em Ciência da Computação, No. 30/05 ISSN: 0103-9741
Editor: Prof. Carlos José Pereira de Lucena Agosto, 2005

Governing Agent Interaction in Open Multi-Agent Systems[1]

Rodrigo Paes, Carlos Lucena and Paulo Alencar[1]

[1] Computer Science Group - University of Waterloo

{rbp,lucena}@inf.puc-rio.br, palencar@csg.uwaterloo.ca

Abstract. Open multi-agent systems are increasingly gaining importance in both academy and industry. These systems are composed of autonomous distributed agents that can be developed by independent teams. Furthermore, agents can have different architectures, and even conflicting goals. However, in some applications domains such as electronic commerce, the system as a whole should inspire confidence in their users and also in the parts involved in the business. Confidence means basically that the system will act according to what is expected, that is, it is behaving according to specifications while maintaining the quality that its services are being provided. However, in an open multi-agent system scenario, the code of the agents are frequently inaccessible, or even the agents themselves are unknown beforehand. Achieve higher degrees of confidence claims for mechanism to monitor system execution and verify if the actual behavior is compatible with what is specified. This paper uses a law-enforcement approach and presents an object-oriented framework to monitor and verify conformity of agents' behavior. First it is used a model to specify agent's interaction. This model is extended to increase its expressivity. Then, based on this extended model the object-oriented framework is built. The main goal of this paper is to share ideas about how a law governing approach can be implemented in a way that it could serve as 'how-to-do-it' models for future work in the same field.

Keywords: multi-agent systems, organizations, interaction protocols, open systems, laws

Resumo. Cada vez mais os sistemas multi-agentes abertos estão ganhando importância tanto na academia quanto na indústria. Estes sistemas são compostos de agentes autônomos distribuídos que podem ter sido desenvolvidos por desenvolvedores diferentes. Além disso, agentes podem possuir arquiteturas diferentes e até mesmo objetivos conflitantes. Entretanto, em alguns domínios de aplicações tal como comércio eletrônico, o sistema como um todo deve inspirar confiabilidade em seus usuários e também nas partes que estão envolvidas com nas negociações. Confiabilidade significa que o sistema irá agir de acordo com o que é esperado, isto é, ele está se comportando de

[1]This work has been sponsored by Ministério de Ciência e Tecnologia da Presidência da República Federativa do Brasil.

acordo com as especificações ao mesmo tempo em que mantém a qualidade dos serviços que ele provê. Em um sistema multi-agentes aberto, o código dos agentes é frequentemente inacessível e mesmo os agentes podem ser desconhecidos a priori. Um mecanismo que monitora a execução e verifica se o comportamento do sistema é compatível com o comportamento especificado pode ser utilizado para alcançar maiores níveis de confiabilidade. Neste artigo, apresenta-se uma abordagem baseada em leis implementada através de um framework orientado a objetos para verificar a conformidade do comportamento dos agentes em relação a especificação.

Palavras-chave: sistemas multi-agentes, organizações, protocolos de interação, leis

In charge for publications:
Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC-Rio Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22453-900 Rio de Janeiro RJ Brasil
Tel. +55 21 3114-1516 Fax: +55 21 3114-1530
E-mail: bib-di@inf.puc-rio.br
Web site: http://bib-di.inf.puc-rio.br/techreports/

1 Introduction

Software has evolved from controllable production environments where the whole system is under control of one team, and so, artifacts can be inspected and their behavior can be mostly predicted, to distributed development environments where developers are frequently unknown beforehand.

Nowadays, software may contain dynamically interacting components, each with their own thread of control, and engaging in complex coordination protocols. These systems are typically more complex to correctly and efficiently engineer than those that simply compute a function of some input through a single thread of control [1].

This trend has led to software systems that have no centralized control and that are composed of autonomous entities. These entities may enter and leave the environment at their will, and they may even have conflicting interests. Multi-agent auction systems are examples of such open and distributed applications [2] [3].

Further, open systems need to rely on critical infrastructures that constitute the backbone for the delivery of their essential services. This is not only because open systems are more prone for overloads, attacks or failures, but also because they need to deal with uncertainty. While open system components are often autonomous, they behave unpredictably when unforeseen situations arise. Taming this uncertainty is a key issue for dependable open software development.

Current techniques to tame this uncertainty focus on predicting the behavior of the components that compose a system, for example using unit tests or model checking. As in open systems the internal aspects of the components are frequently unaccessible, these technics become insufficient. There is a need for specifying and verifying the observable behavior of the components, i.e., how components behavior in face of the execution environment or in face of other components.

Our approach tackles with the problem by specifying, monitoring and enforcing the interaction of components that compose an open distributed system, where components are seen as agents. The conventional approach is to hard-code the interaction specification into all members of a multi-agent system. Separating the specification from the agents facilitates the dynamic change of behavior of a distributed system, and also enables reuse of an expert's knowledge. More specifically, we refer to a model which provides the concepts required to specify the interaction among agents. Based on this model, it is presented an object oriented framework that can be used to monitor and enforce agents' interaction.

The rest of this paper is organized as follows. Section 2 presents the model of interactions used to specify agents' interactions. This model is extended with concepts of constraints and actions. Section 3 presents the object-oriented framework, which is designed based onn the extended model. In this section, it is given design decisions and some examples. Section 4 shows an example of application modeled using the concepts of the interaction model, specified using a declarative language, called XMLaw, and finally, how the framework behaves in face of a specific interaction scene. Section 5 presents some works that are in some sense related to this paper. Finnaly, we depict some discussions about the current paper and ongoing resarch in Section 6.

2 Modelling Interactions

The model presented in this section is used to represent interactions in an open distributed environment. Basically, interactions should be analyzed, and after that, described using the concepts proposed in the model. Then, the concepts are mapped to a declarative language, called XMLaw[2].

[2]XMLaw can be seen in more details in [4].

Interaction's definitions are interpreted by a software framework that monitors components' interaction and enforces the behavior specified on the language.

Once interaction is specified and enforced, despite the autonomy of the agents, the system's global behavior is better controlled and predicted. Interaction specification of a system is also called laws of a system. This is because besides the idea of specification itself, interactions are monitored and enforced. Then, they act as laws in the sense that describe what can be done, what cannot be done and what should be done. Bellow, we present each concept and their related representation in XMLaw.

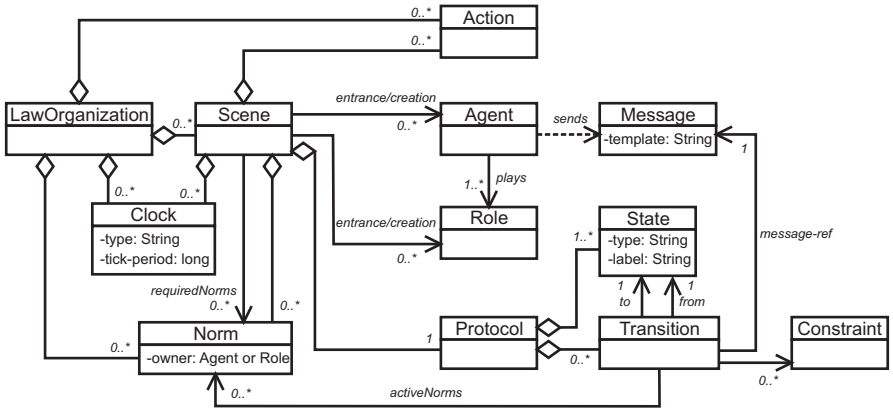

Figure 1: Model of Interaction

This model provides computational concepts that allow specify interaction laws in multi-agent systems. Most of this model was already reported in [4] and therefore the concepts are very briefly introduced in this paper to make it self-contained. However two novels abstractions (actions and constraints) are introduced in the current version and they are detailed in sections 2.1 and 2.2.

The outer concept of this model is the LawOrganization. We can see this element as representing the interaction laws (or normative dimension [5]) of a multi-agent organization. A LawOrganization is composed of scenes, clocks, norms and actions. Scenes are interaction contexts that can happen in an organization. They allow to modularize interaction breaking the interaction of the whole system in smaller parts. Clocks introduce global times which are shared by all scenes. Norms capture notions of permissions, obligations and prohibitions regarding agents' interaction behavior. Actions can be viewed as a consequence of any interaction condition, for example, if an agent acquires an obligation, then the action 'A' should be executed; if an agent that is critical for the system is overloaded, new replicas for this agent should be created [6].

Scenes define an interaction protocol (from a global point of view), a set of norms and clocks that are only valid in the context of the scene. Furthermore, scenes also identify which agents are allowed to start or participate of the scene. Non-deterministic automatons, enhanced by the constraint element and by the possibility of integration with the other elements from the model, represent interaction protocols. Constraints are used as conditional firing of protocol transitions allowing to check complex conditions based on the message contents.

Figure 1 can be viewed as a structural and static view from the concepts used to specify interactions. These elements relate to each other through the associations, dependencies and aggregations relationships showed in this figure. However, to achieve lower coupling levels among these elements, we also provide a dynamic model based on events that reify the relationships of the conceptual model in a flexible way.

Events are the basis of the communication among law elements, that is, law elements dynamically relate with other elements through event notifications. Basically, we can understand the dynamic of the elements as a chain of causes and consequences, where an event can activate a law element; this law element could generate other events and so on. For example, the arrival of a message generates an event (*message arrival*), this event may activate a transition (*transition activation*), transition in its turn may activate a clock (*clock activation*), which generates a (*clock_tick*) event, and lastly it activates a norm (*norm activation*). Figure 2 shows this chain of causes and consequences, Figure 3 shows a list of all events generated by the elements from the model, and Figure 4 summarizes all law elements that can generate and sense events.

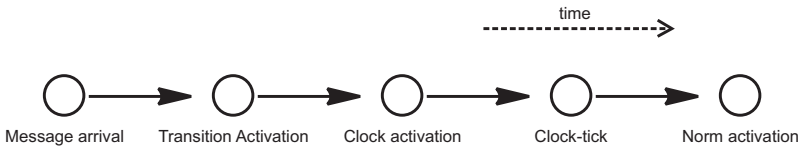

Figure 2: Chain of Events

Message	Description
message_arrival	Message arrives at mediator
transition_activation	Transition is fired
clock_tick	Time specified in a activated clock has ellapsed
deactivation_event	Generic event for any deactivation
norm_activation	Norm has been activated
clock_activation	Clock has been activated
final_state_reached	A final state in a protocol has been reached
time_to_live_elapsed	Scene's time to live has been elapsed
successful_scene_completion	Scene has been successful completed
failure_scene_completion	Scene has been completed with failure
action_activation	Action has been called

Figure 3: Event Types

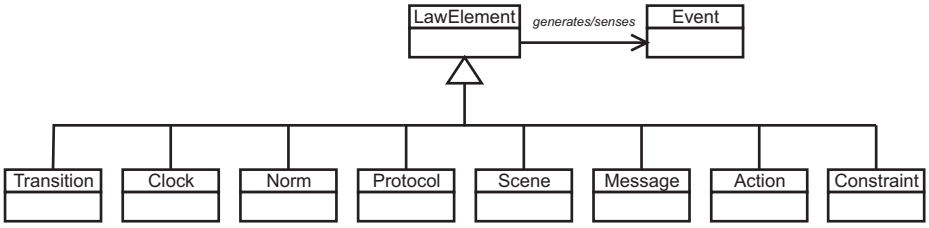

Figure 4: Law Elements that Generate and Sense Events

2.1 Constraints

Constraints concern verification on message values. Messages carry information that are verified in many ways; the attribute *message pattern* (Section 3.2.2) verifies the shape of messages, describing the required pattern. However, this attribute does not describe what the allowed values are on certain parts of messages. For example, a *message pattern* for the content field could be:

content(television, brand(AnyBrand), price(Amount)). Then, a certain application requires that the value of *Amount* variable has to be between 50 and 100. Using only the *message pattern* this requirement can not be achieved. *Constraint* element tackles just this point providing a flexible way to perform this kind of requirement.

Constraints are specified using a component that implements the functionality. Current implementation of the model uses Java components to implement constraints. In this way, developers are free to develop constraints as complex as needed, only bounded by Java limitations.

Although Java enables definition of very expressive constrains, using Java or any other non-declarative language brings a drawback, it separates the meaning of the constraint from the interaction specification, putting it in the java code. If there were no mechanism to externalize the meaning, would be necessary inspect the java code to extract any meaning. For this reason, constraints have an attribute named *Semantics* that allows specify the constraint's semantic. There is no restriction to the formalism used on semantic's specification.

In short, constraints acts on message values and have a semantic field. However, once transitions have only a reference to messages, the same message can be reused in many different transitions. Then, despite constraints act on messages, constraints are not coupled to messages. They are coupled to transitions and acts on the message that activates the transition. XML in Listing 1 shows how it is done. The attribute *class* refers to a Java component that implements the restriction.

```
<Transition id=''a-transition-id'' from=''from-state''
to=''to-state'' message-ref=''rfq''>
    <Constraints>
        <Constraint class=''ajavapackage.AConstraintClass''>
            <Semantic>Semantic for this constraint</Semantic>
        </Constraint>
    </Constraints>
</Transition>
```

Listing 1: Constraint Acting on Message "rfq"

2.2 Action

A very important function of nowadays software is to have the skill to dynamically self-adapt in response to changes on available resources, user needs or system faults [7]. There is an increasing demand for identifying and solving the problems even in highly dynamic and unpredictable environments. Automated self-healing has appeared as solution to these problems. Automated self-healing can be defined as the skill of a system to automatically detect, diagnose and recover software and hardware problems [8].

Action is the element from the model intended to perform automated self-healing. In fact, actions are domain-specific java components that run tightly integrated to interaction's specification. Then, as most of other elements, actions can be activated by transitions, norms, clocks or even other actions.

The self-healing activity done by actions can be viewed in three phases: monitoring, detection and healing. Monitoring is done through interaction's specification, i.e., certain software conditions can be monitored. For example, detecting that an agent is taking too long to communicate. Detection phase occurs when the condition being monitored happens and the action is called. Finally, the healing activity is performed by the implementation of the action component. Figure 5 shows these activities.

The declarative specification of actions is done as shown in Listing 2. The *class* attribute specifies the java component in charge of performing the healing. The <*Activation*> tag contains

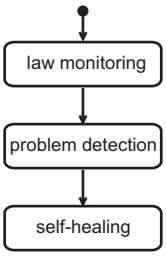

Figure 5: Action Activities

references to other elements that can activate the action.

```
<Actions>
  <Action id="anActionId" class="apackage.ActionClass">
    <Semantics>Semantics for this action</Semantics>
    <Activation>
        <Element ref="generatorReference" event-type="type"/>
    </Activation>
  </Action>
</Actions>
```

Listing 2: Action in XMLaw

3 Object Oriented Framework for Monitoring and Enforcement

Advances on agent technology relies on development of models, mechanisms and tools to build high quality systems. Design and implementation of such systems are still expensive and very error prone. Software frameworks deal with this complexity reifying proven software designs and implementations in order to reduce the cost and improve the quality of software. In this way, a framework is a reusable, semi-complete application that can be specialized to produce custom applications [9].

In this section, we present a framework that supports development of open distributed systems providing compliance with both the model of interactions proposed previously and the declarative language XMLaw.

The framework has a set o modules that supports three types of users:

- "Law developer" represents the developer responsible for specifying the laws. Law developers must understand the application under construction; know the law concepts; and then, specify the laws for the application.
- "Agent developer" represents the developer responsible for building the agents of a multi-agent system. Those developers know about the existence of the laws and should design the agents in compliance with them.
- "Software infrastructure developer" deals with law enforcement software support. Sometimes there is no software infrastructure that implements the law enforcement mechanism or the existent infrastructures are not suitable for the systems that are under development. In these cases, software infrastructure developers should modify the existent infrastructure, or even construct a new one.

The framework provides support for each one of those developers. First, it is provided a declarative language called XMLaw for law developers, which allows the specification and maintenance of the interactions of multi-agent systems. XMLaw specifications are interpreted and enforced by the law enforcement framework. The framework provides this software support, which contains a number of hotspots that can be extended by software infrastructure developers. Lastly, agent developers are provided with classes and interfaces that support building agents integrated to the law enforcement software.

The framework is composed of many modules, which are illustrated in Figure 6. This figure also highlights which modules are hotspots. Finally, Figure 7 shows which modules are related to each kind of developer. Details of each module such as extension points and usage mode are provided in the further sections.

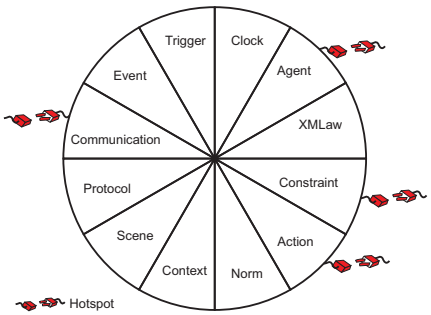

Figure 6: Framework Modules

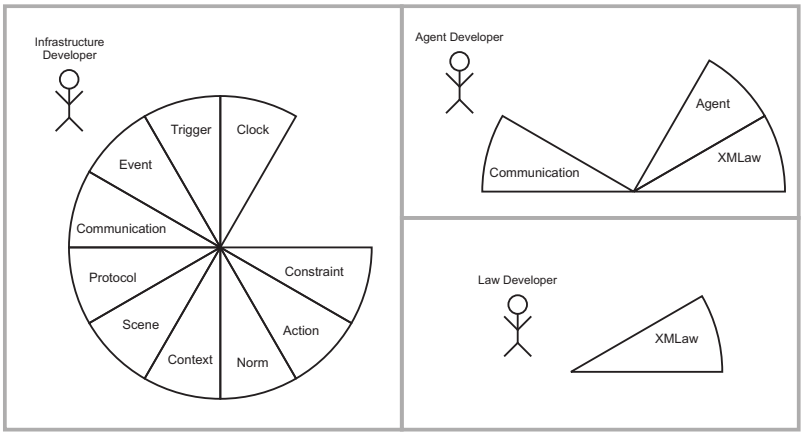

Figure 7: Software Infrastructure Developer View of the Framework

3.1 Interaction Model

In open systems, one piece of available information about agents is their observable behavior through message exchanges. The law enforcement mechanism should intercept these messages, and then, enforce the desired system behavior. This mechanism acts as a mediator among agents. This mediator is not necessarily neither one unique entity or centralized. If the mediator is centralized, some degree of scalability can be achieved using a pool of mediators. However, in this case, we still having a unique point of failure. Another solution is to think of mediator as many decentralized entities that monitor the system execution. In this sense, depending on how mediators are implemented (centralized or decentralized), higher or lower level of scalability can be achieved. This mediator idea was already reported, with minor variations, in a number of publications such as controllers [10], governors [11], and inter agents [12].

The law enforcement mechanism life cycle consists of three high-level activities (Figure 8): (i) interception; (ii) enforcement; and (iii) redirection. In the first stage, the mechanism intercepts messages exchanged between agents. Then, the mechanism interprets the law specifications and check if the message complies with the laws. Finally, the mechanism redirects the message to the real addressee. In this sense, the observable behavior of agents can be controlled and the laws can be enforced.

The framework presented follows the architecture presented in this section. In the next sections, it is detailed the main modules that compose the framework.

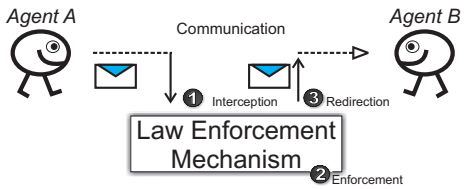

Figure 8: Framework Interaction Architecture

3.2 Communication Module

The communication module is composed of two sub modules: communication layer and message module. The former is concerned with heterogeneity questions regarded to agent communication and the latter describe what kind of message agents have to use in order to communicate.

3.2.1 Communication Layer

Agents can communicate using different ways; they can use SOAP [13] based communication, implement their own communication standard using sockets, or use a more accepted agent standard such as defined by FIPA [14]. Each application has different requirements for communication mechanisms; some of them focus on performance issues, flexibility, interoperability, among others. The communication layer is concerned with providing an interface that allows domain-specific applications to change the communication mechanism when it is needed.

The interface *ICommunication* defines methods for sending and receiving messages. Concrete communication mechanisms should implement this interface in order to provide methods' functionalities. The *send(Message msg):void* method sends the message to the addressees specified. The *waitForMessage():Message* blocks the execution until a message arrives, and when it does, the message is returned. The *waitForMessage(long milliseconds)* method blocks the execution until a message arrives or until the time specified in the parameter has elapsed, and returns the just arrived message or *null* in case of no message has arrived. This interface defines one more method named *nextMessage():Message*, but despite of the *waitForMessages* methods this method does not block the execution. The expected behavior of this method is to return the next unread message if it exists, or *null* otherwise. This kind of behavior suggests that implementers of *ICommunication* interface should use some kind of queue to keep all received messages by an agent, and when the *nextMessage* is invoked, it returns the first message of the queue.

Figure 9 shows the class diagram for communication layer. Due to its methods definitions, the interface *ICommunication* has a dependency relationship with the class *Message*. Moreover, this figure also shows how we implemented this interface using the JADE framework [15]. JADE implements a FIPA compliant communication mechanism. This mechanism is encapsulated in the *jade.core.Agent* class, provided by JADE. Then, *JadeCommunication* class reuses the implementation of this mechanism delegating the methods from the *ICommunication* interface to an instance of *jade.core.Agent* class. Listing 3 shows a piece of code of the *JadeCommunication* class. This class has an attributed named *jadeAgent*, instance of *jade.core.Agent*, which is initialized when a *JadeCommunication* instance is created. The *send* method first transform the framework message format to JADE specific message format, and delegates the request to the *jadeAgent* instance.

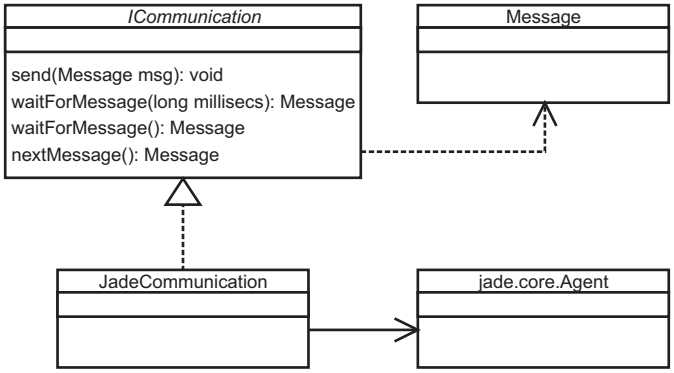

Figure 9: Communication Layer

```
public class JadeCommunication implements ICommunication{
    ...
    protected Agent jadeAgent;

    ...

    public JadeCommunication(String name) {
        ...
        jadeAgent = new Agent();
        ...
    }

    ...

    public void send(Message msg) {
        ACLMessage message = this.messageWrapper.transform(msg);
        jadeAgent.send(message);
    }
    ...
}
```

Listing 3: Code Fragment for the Communication Layer using JADE

3.2.2 Message Module

Messages are a very important part of the law enforcement approach. Law enforcement is concerned with enforcing agents' behavior from interaction standpoint, and agents interact by exchanging messages[3]. Therefore, the enforcing is made through the analysis of exchanged messages. The *Message* class represents the messages exchanged among agents.

The *Message* class contains many pre-specified fields needed for an effective agent communication. These fields are specified in a pair-value format. Although the number of fields used might vary, there are five mandatory fields: *performative*, *sender identification*, *sender role*, *receiver identification* and *receiver role*. The *performative* contains the communicative act that the sender agent wants to communicate. *Sender* and *receiver identifications* are concerned with the identification of sender and receiver agents respectively. *Sender* and *receiver roles* identify the roles that sender and receiver agents are playing during a conversation. Other pre-specified fields include *conversation identification*, *content* and *scene authorization*. *Conversation identification* allows identify contexts of conversation. Agents may keep the tracking of all exchanged messages in a conversation using the *conversation identification*, or *conversation id* for short. The *content* is the

[3]Despite other communication forms used by agents, such as environment based communication, in this paper only message exchange is considered.

information that agents send to other agents. At last, *scene authorization* represents the authorization agents must have in order to interact with other agents in a certain scene. This authorization is issued by the mediator agent, and it can be requested through the *Mediator Protocol* (Section 3.4.2).

XMLaw defines an element named *message-pattern*. This element is concerned with how pattern of messages can be recognized. The framework performs the pattern recognition using both the *format()* method of the *Message* class and the class *MessagePattern*. The *format()* method returns the message in a text-based formatting as shown in Listing 4.

```
message(performative,
        sender(sender identification, sender role),
        receiver(receiver identification, receiver role),
        content(content) ).
```

Listing 4: Returned Value of the *format()* Method

The *MessagePattern* class manages the creation and destruction of *IMessagePatternMatcher* interface implementors. Such interface defines only one method, as shown in Listing 5.

```
match(String formattedMessage, String template): Hashtable
```

Listing 5: Interface *IMessagePatternMatcher*'s method

This method contains two parameters, the *formattedMessage* resulting of *format()* method's invocation, and the template that the *IMessagePatternMatcher* should verify if the *formattedMessage* matches. If the message matches the pattern and if the pattern allows the use of variables, this method should return a *Hashtable* containing all the variable bindings.

One alternative for implementing this approach is implementing a *Prolog* like message pattern. For example, in *Prolog* variables are represented capitalizing the first letter, then a possible pattern would be *price(Value)*, where *Value* is a variable. Then, an invocation for the match method receiving *match("price($34,45)", "price (Value)")* should return a hashtable containing the pair *["Value","$34,45"]*. However, other formats of messages can be used just providing different implementors for the *IMessagePatternMatcher* interface.

To summarize, message patterns are specified in XMLaw, those patterns must be understood as a concrete implementation of *IMessagePatternMatcher*, and the framework is in charge of orchestrating all these method invocations. Figure 10 shows the classes that compose the message module and their relationships.

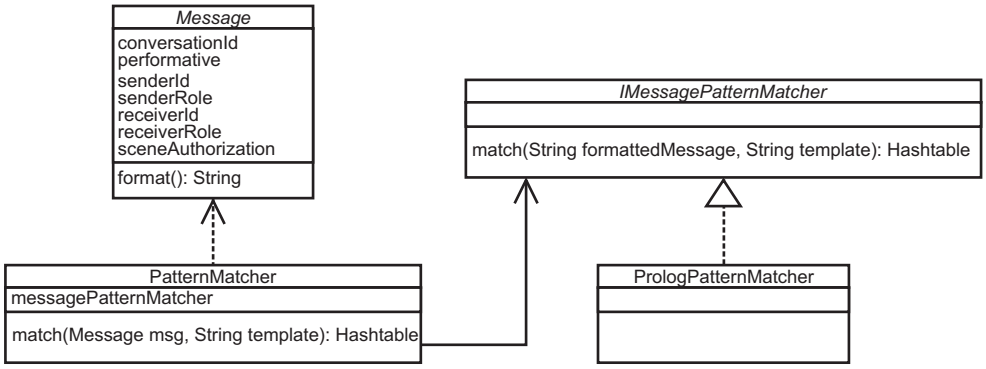

Figure 10: Message Module

3.3 Mediator Agent

Most of the framework is implemented as mediator agent modules. The mediator agent monitors all interactions and makes sure that interactions are compliant with the specifications. The mediator performs a number of activities that are depicted in Figure 11. First, the mediator waits for receiving messages. Once a message arrived, it checks if the message belongs to the mediator protocol. If it does, the mediator proceeds with the protocol execution. Otherwise, if the message belongs to some agent conversation, the mediator starts the process of enforcing, and if the laws allow, the message is redirected to the addressee agent. This sequence of activities is repeated while the mediator agent is running.

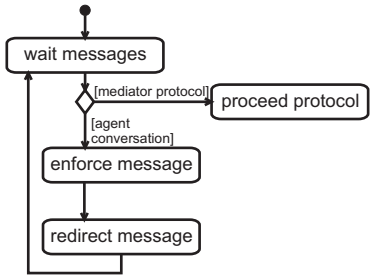

Figure 11: Mediator Activities

Executing these activities requires that the mediator agent has ability to do things such as interpret the laws, and since the law enforcement approach is still evolving, this agent should be flexible and expansible to accommodate future changes. In this way, the mediator agent is based on nine main modules where each module provides a specific functionality. These modules are described in the next subsections.

3.3.1 Events

The communication among the modules is mainly based on event notifications. This approach leads to a low coupling level among modules and also leads to more flexible system designs [16]. In event-based systems, there is a module, named Event Manager, which provides an interface containing all operations needed for enabling event-based communication. This interface contains the abstract operations *fireEvent*, *subscribe* and *unsubscribe*. The *fireEvent* operation is called by clients named *Producers*, and this operation notifies all subscribers about the event occurrence. *Producers* generate events in the event system. *Subscribers* registry their interest in certain kind of events through the *subscribe* operation. Calling the *unsubscribe* operation undo the *subscribe* operation.

The framework implements an event-based system through the collaboration of several classes and interfaces. The *IEvent* interface defines the basic behavior of all events (Figure 12). This interface extends *IIdentifiable*, which means that all events have identifications. Moreover, *IEvent* defines three methods:

- *getType():int* - returns the type of the event. Although events could have been identified only by their classes, using integers to identify events allows a faster event notification implementation through binary arithmetic, and also allows more uncoupled modules once modules have to know only the type of the event, unlike knowing the class. The class *Masks* has constants that identify all types of events contained in the framework.

- *getInfo():InfoCarrier* - Events carry data that might be important for subscribers. The data is encapsulated in an *InfoCarrier* object. This object is a table where information can be stored and retrieved. The method *getInfo* returns the object *InfoCarrier* contained in this event.

- *getEventGenerator(): IIdentifiable* - Conceptually, events are generated by *Producers*. In the framework, producers can be any *IIdentifiable* object. This method returns the producer of this event.

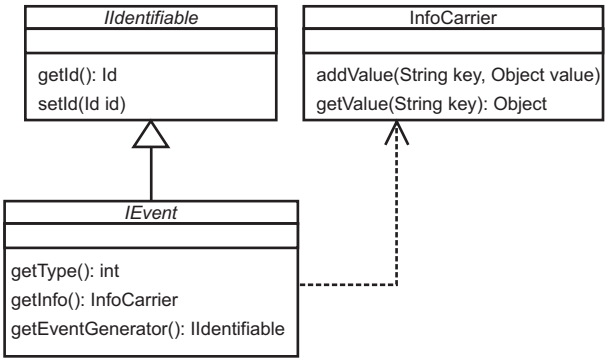

Figure 12: IEvent Interface

The interfaces *ISubject* and *IObserver* define respectively the behavior of *Event Managers* and *Subscribers*. Then, subscribers registry interest in certain type of events using the method *attachObserver*. This method has two parameters, the former is the observer (or subscriber) itself, and the latter is a mask related to the type of events that the observer is interested in. The mask can be formed using as many event types as required. This mask is built using the binary operator 'I'(OR) and constants defined in the class *Masks*. Then, for instance, if an observer is interested in clock and norm activations, it should call:

attachObserver(this, Masks.CLOCK_ACTIVATION | Masks.NORM_ACTIVATION);

In the framework, there are many implementations for the interface *IObserver* such as *Transitions* and *Clocks*. They are detailed in further sections of this paper. However, the framework only provides one implementation for *ISubject* interface. This implementation is provided by the class *Subject*. This class has a particular implementation for the *fireEvent* operation. Usual implementations run a while loop notifying all observers about the event occurrence, such as shown in Listing 6.

```
... for (int i = 0; i < observers.size(); i++) {
    IObserver anObserver = (IObserver) observer.get(i);
    anObserver.update(event);
} ...
```

Listing 6: Usual Implementation for the *fireEvent* Method

However, this implementation has some drawbacks. From the caller of the method *fireEvent* point of view, it may wait for too long if there are computer intensive tasks on some observers' update methods. Unexpected behavior may also happens, once implementations of the update method can generate other events. It means that events generated latter can arrive at observers first than events generated earlier.

For example, Figure 13 shows a sequence diagram where a client produces the event *A*. There are two observers registered in the subject, the *observer1* is interested in events *A*s, and the *observer2* is interested both in events *A*s and *B*s. Then, the subject first notifies the *observer1* of the occurrence of *A*. However, the *observer1*'s *update* method produces a *B* event. Then the subject checks that only the *observer2* is interested in this type of event, and it notifies the *observer2* about the occurrence of *B*. The update's implementation for the *observer2* requires that *A* has occurred before *B* in order to execute the method *operation()*. Since *observer2* is first notified about *B*, *operation()* is not called. But, in fact, *A* happened before *B*, and therefore, *operation()* should have been called.

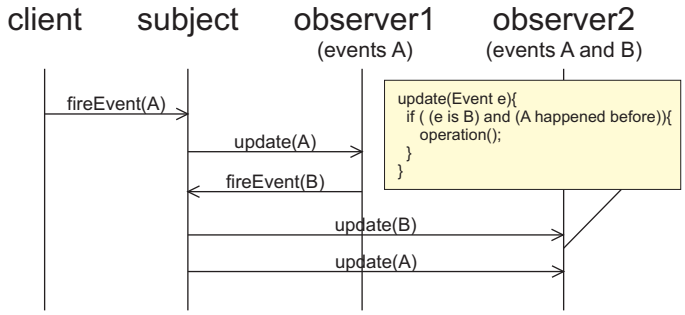

Figure 13: Problem with the Usual Implementation of Events Notification

In order to avoid this problem, the subject provided by the framework uses a different strategy. When clients produce events, the *Subject* adds the event to be generated in a queue. The *Subject* extends the java class *Thread* and overwrite the method *run()*. It means that *Subject* runs in a different thread of execution. The pseudo code for the *run()* implementation is shown in Listing 7. This method contains a loop where it is constantly verified if there are events in the queue. Events are put in the queue through the method *fireEvent*. This technique guaranties that the observers will be notified about event occurrence on the correct order, and the callers do not have to wait for long event if it exists computer intensive update's implementations. The Figure 14 shows the class diagram for the classes related to *ISubject* and *IObserver*.

```
while (true){
    while (events.size()>0){
        IEvent event = (IEvent)events.pop();
        Vector observers = get all observers for this event;
        for (int i = 0; i < observers.size(); i++) {
            IObserver anObserver = (IObserver) observers.get(i);
            anObserver.update(event);
        }
    }
    wait();
}
```

Listing 7: Subject's *run()* Method

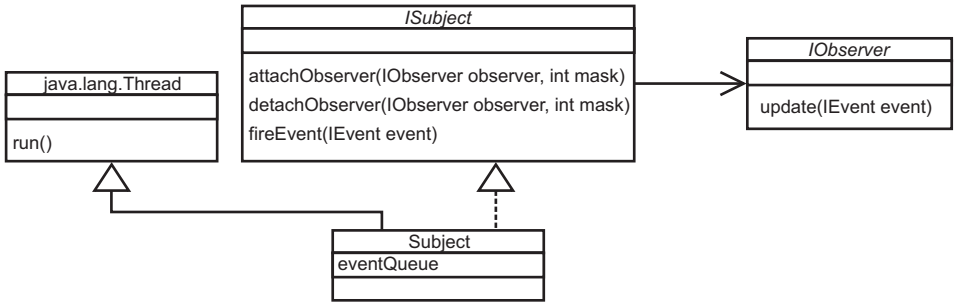

Figure 14: Observer and Subject

3.3.2 Trigger

Triggers are software components that may be activated by events if certain conditions are satisfied, and once activated they execute some action. This idea is similar to Event-Condition-Action (ECA) systems, and it is largely used in database systems [17]. The framework contains a trigger module that specifies a set of classes and interfaces, which are used for implementing this idea.

In fact, as the communication among framework's modules is mainly based on event notifications, the trigger module acts as an observer (or consumer) of generated events in order to activate or deactivate triggers. Many important concepts of the model, such as Clocks and Norms, are implemented as triggers. It brings as benefits reuse and reliability, once trigger module is used and tested by many other modules.

A trigger is represented by the interface *ITrigger*. This interface contains methods for adding conditions that activate and deactivate a trigger. Those conditions are expressed as an occurrence of types of events generated by some producer. That is because many different producers may generate the same kind of event, then specifying both producer and type of event makes possible to specify that only a specific producer activates the trigger.

Triggers are able to generate events indicating their activation. The method *getActivationEvent* returns the event that is generated when the trigger is activated. According with the ECA model discussed earlier, when a trigger is activated an action should be executed. In this framework, trigger actions are represented by the abstract class *TriggerFiring*, and triggers have the method *getTriggerFiring* that returns an instance of *TriggerFiring* class. Then, for instance, once a clock (*ITrigger*) is activated, it creates a *TriggerFiring* that starts counting the time.

Once created, *TriggerFiring*s are not running yet. Basically, they can be in one of these three states: *created*, *running* and *stopped*. The *created* state occurs just after the instantiation. *TriggerFiring*s enter into *running* state when their method *start()* is called, and they only leave this state and go into *stopped* state when the method *stop()* is invoked.

There is a component in charge of controlling when to call the *ITrigger*'s methods, such as *getTriggerFiring*, and as well as managing *TriggerFirings*' life cycle. The *TriggerManager* is the framework's class in charge of those responsibilities. It can be viewed on Figure 15, which depicts the class diagram of trigger module.

TriggerManager keeps the list of all Triggers. It also implements the *IObserver* interface, and therefore, it listens for all events that happen in the framework. In its update's method, the *TriggerManager* controls the activation and deactivation of triggers. Listing 8 shows the idea of triggers activation. Basically, when an event happens, for each trigger, it is verified if the trigger is activated by the event. If it is, then the trigger is requested to return the action as consequence of its activation, *TriggerFiring*. The action is then started, and an event related to the trigger activation is scheduled to be thrown. The deactivation of triggers occur in a similar way, as illustrated in Listing 9.

```
for (int i = 0; i < triggers.size(); i++) {
    ITrigger trigger = (ITrigger) triggers.elementAt(i);
    if (trigger.isActivatedBy(event)){
        TriggerFiring activeTrigger = trigger.getTriggerFiring(info);
        enabledElements.add(activeTrigger);
        activeTrigger.start();
        IEvent eventToThrow = trigger.getActivationEvent(info); // fires it latter
        eventsToThrowList.add(eventToThrow);
    }
}
```

Listing 8: Triggers Activation

```
for (int i = 0; i < enabledElements.size(); i++) {
    TriggerFiring enabledEvent = (TriggerFiring) enabledElements.elementAt(i);
    if (enabledEvent.isDeactivatedBy(event)){
        this.enabledElements.remove(enabledEvent);
        enabledEvent.stop();
        IEvent eventToThrow = enabledEvent.getDeactivationEvent(info);
        eventsToThrowList.add(eventToThrow);
    }
}
```

Listing 9: Triggers Deactivation

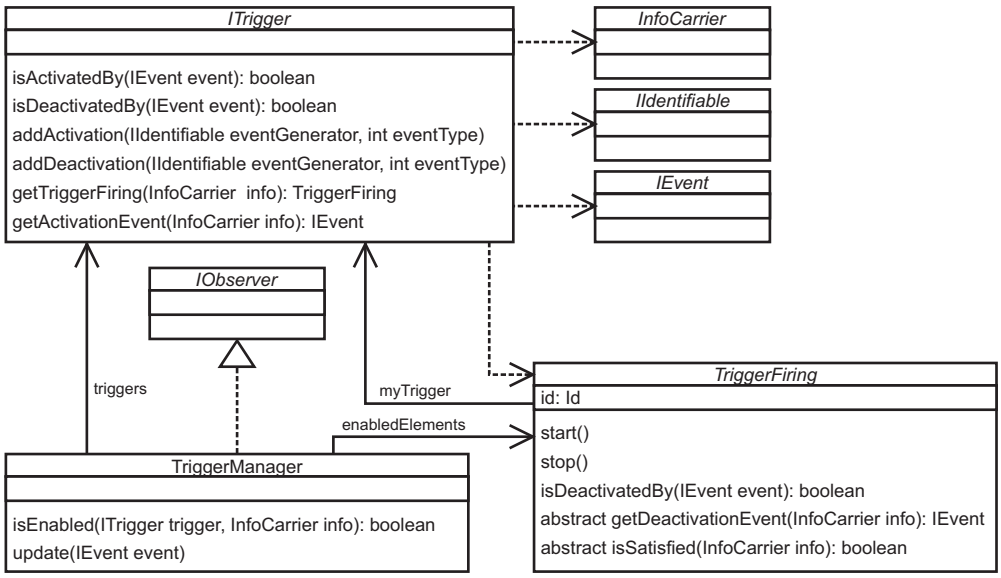

Figure 15: Trigger Module

3.3.3 Contexts

Contexts are usually hierarchic and limit the visibility of actions and information. The idea is similar to a file system structure. In a file system, each directory may contain a number of files and other directories. In this way, each directory provides a context for the files contained in it.

A law specification may be composed of law definitions for many organizations, and each organization's law is composed of scenes. Those compositions define contexts, where law elements defined in an organization context are visible to all scenes, but elements defined in a scene context are only visible to the scene.

In the framework, contexts limit the visibility of events and triggers. Each scene instance has an associated context, which is child of an organization context. Then, whatever happens in a scene instance is visible both to the scene instance itself and to the organization which the scene instance belongs to. Due to this visibility schema, its is possible that events inside scenes such as transitions activate clocks specified at the organization context. It is also possible that scenes use, for example, norms at the organization context as prerequisite to transition activations.

The *Context* class represents contexts, and its structure is shown in Figure 16. Since contexts limit visibility of events and triggers, the class *Context* reuse the implementation of classes *Subject* and *TriggerManager*. A context encapsulates those classes in the way that scenes, organizations and other elements have to know just the *Context* class, to generate and receive events. The *Context* class delegates the implementation of the methods *attachObserver*, *detachObserver* and *fireEvent* to the class *Subject*, that acts as an event manager; and it also delegates the implementation of the methods *isEnabled* and *addTrigger* to the *TriggerManager* class.

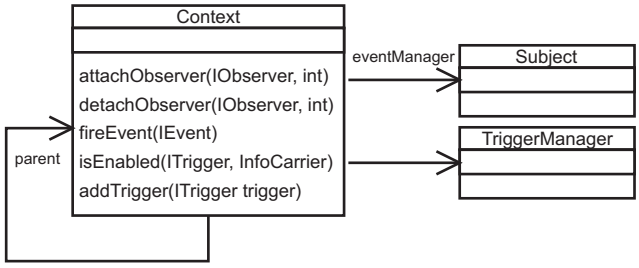

Figure 16: *Context* Class

Context classes have an attribute called *parent* that references the parent context if it exists or *null* in the case of being an organization context (root). When an event is generated in a *Context* object, the event is also propagated to its parent context. Furthermore, the verification if a certain trigger is active or not should be done also in the parent context, in case of the trigger is not active. Those propagations are implemented in the methods *fireEvent* and *isEnabled*, and Listing 10 shows a pseudo-implementation.

```
public boolean isEnabled(ITrigger trigger, InfoCarrier info) {
    if (!triggerManager.isEnabled(trigger,info)){
        if (parentContext!=null){
            return parentContext.isEnabled(trigger,info);
        }else{
            return false;
        }
    }
    return true;
}

public void fireEvent(IEvent event) {
    eventManager.fireEvent(event);
    if (parentContext!=null){
        parentContext.fireEvent(event);
```

```
    }
}
```

Listing 10: Event Propagation and Trigger Verification

3.3.4 Clocks

Clocks are implemented extending trigger and event modules. The *Clock* class represents a *Clock* element from conceptual model and extends the interface *ITrigger*. Clocks generate instances of *ActivatedClock* in response to callings of its *getTriggerFiring* method (inherited from *ITrigger* interface). *ActivatedClock* extends *Runnable* interface and, therefore, runs as a separated thread of execution. Once started by *TriggerManager*s, an *ActivatedClock* instance begins to count the time, and when the time elapsed is equal to its *timeout* attribute, it generates a *ClockTick* event. Figure 17 shows the class diagram for the clock module.

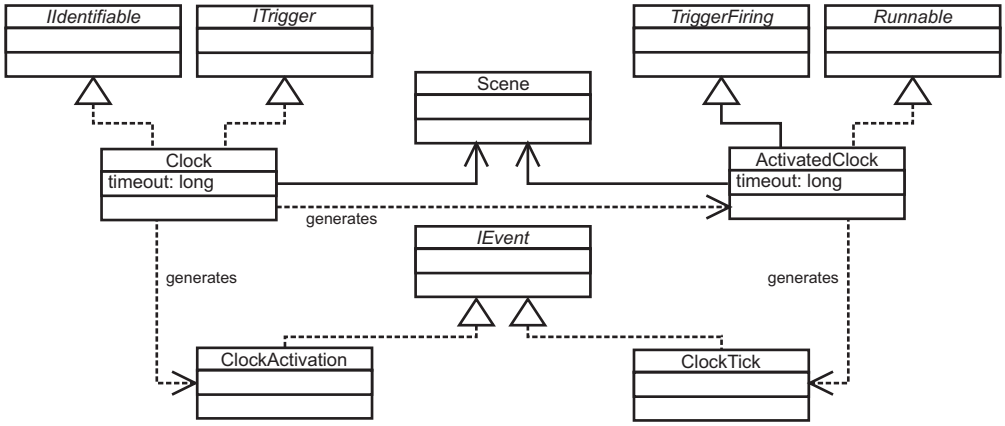

Figure 17: Clock Module

3.3.5 Norms

Norms are represented by the class *Norm*. As the clock module, norms are implemented extending trigger and event modules. Norms are triggers listening for events that activate and deactivate themselves. Once activated, norms generate instances of *NormActivation* and then an activated norm (*ActivatedNorm* class) is created and put in a context. Furthermore, a norm can be of three types: *Obligation*, *Permission* and *Forbidden*. Figure 18 shows those classes.

A very important aspect of the norm module is related to the attributes *ownerVariable* and *ownerValue* in the classes *Norm* and *ActivatedNorm*. Norms, such as a permission are conceded to a specific agent or to a certain agent role. It means that norms allow specify that agent *X* has permission *P*, or every agent playing role *R* has the obligation *O*. The way of obtaining this functionality is implemented through *ownerVariable* and *ownerValue*. The *ownerVariable* specifies who is the owner of a certain norm. This variable should exist in the information carried through the *InfoCarrier* object (Section 3.3.1). Usually the first event in the system is the *MessageArrival*. This event puts in the *InfoCarrier* some information about the message that just arrived. Protocol receives this event and checks if some transition will be activated. The transition in its turn executes the pattern matching and then, it is made a binding of every variable defined in the message

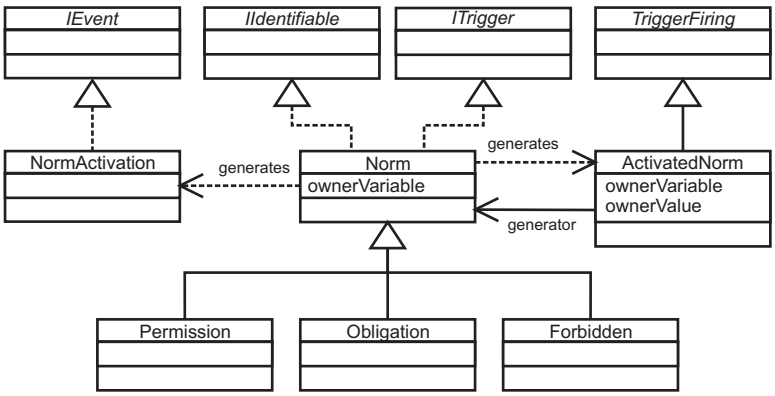

Figure 18: Norms

pattern, and those bindings are put in the *InfoCarrier* object. From this moment on, listeners of transition activation events have access to those bindings, and they also propagate those bindings to further events. When a norm is activated, it looks for the *ownerVariable* in the *InfoCarrier* object and retrieve its value. This pair *ownerVariable* and its value is stored in the class *ActivatedNorm*, which means that a norm was activated to an specific agent or agent role.

3.3.6 Protocol

The protocol module implements a non-deterministic finite state machine [18] where interaction protocol of agents can be specified. The class *Protocol* (Figure 19) represents this machine. *Protocol* implements the *IObserver*[4] interface, and it listens for events of arrival of messages. The *Protocol* class also keeps a reference to the initial state and to a list of all current states.

States are represented by the class *State*, and they have a list of all outgoing transitions. Then, once the protocol receives a message, it executes the algorithm shown in Listing 11. Basically, the protocol controls the list of current states, but the logic of changing of states is delegated to the states themselves. States in turn have their algorithmic shown in Listing 12. This listing shows that states manage the list of next states delegating the responsibility to transitions. In Fact, it is the *Transition* class that "knows" if the protocol should change the state. Transitions execute the activities shown in Figure 20. First, it is verified if the message just arrived has the same pattern as specified in the XMLaw. Then, it is checked if all norms that should be enabled are really enabled. The same is done for the norms that should be disabled. Lastly, the constraints are executed. If all those conditions are satisfied, then the transition is fired and a transition activation event is generated.

```
List futureStates = new List(); for each current state{
    futureStates.add( state.step() );
}

if (futureStates.size == 0){
    sendMessage ( Message not allowed );
}else{
    currentStates <— futureStates;
    redirectMessage;
}
```

Listing 11: Protocol's Pseudo-Code

[4] IObserver is described in Section 3.3.1

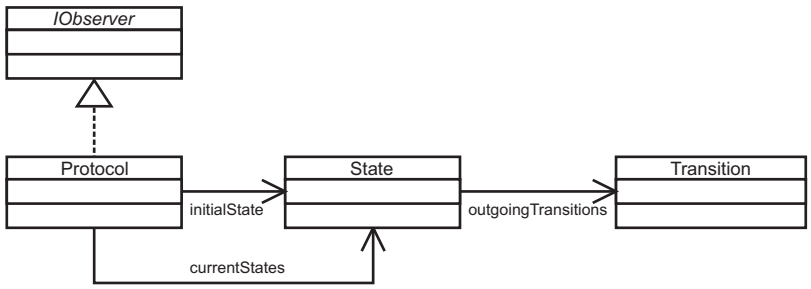

Figure 19: Protocol Module

```
List nextStates = new List(); for each outgoing transition{
    if transition fires{
        nextStates.add( transition.getDestinationState() )
    }
} return nextStates;
```

Listing 12: State's Pseudo-Code

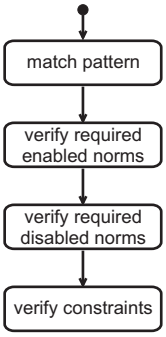

Figure 20: Transition Activities

3.3.7 Scene

Scenes are represented by the *Scene* class. This class defines a context for events, norms, clocks and other law concepts. This class acts as an *IObserver* of its own context, and acts as a *ISubject* delegating the requisitions to its context object. In this way, other classes such as *Norms* and *Clocks* have only to know the *Scene* where they are placed, asking the *Scene* to generate and inform about events.

Figure 21 shows the class diagram for Scenes. Scenes have reference to its interaction protocol (*Procotol* class), to the context that it is defined for the scene, to a list of all required norms in order to begin a scene execution, and to the organization that the scene belongs to. Furthermore, scene's life cycle is implemented through the methods initialization() and finalization(). They are called by the framework when a scene have to be created and destroyed, respectively.

3.3.8 Constraints

Figure 22 shows the class diagram for the constraints module. The interface *IConstraint* defines only one operation: *constrain*. This operation is called by transition objects. It should return *true* if

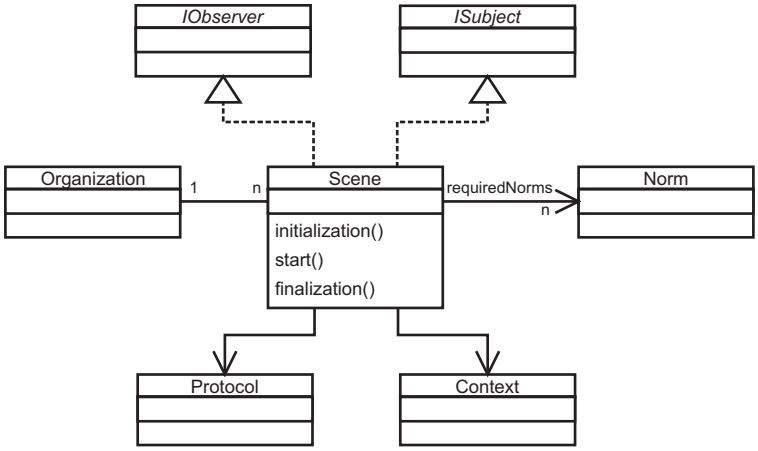

Figure 21: Scene Module

the transition should be constrained by this constraint, and *false* otherwise. Listing 13 shows an example of constraint implementation. This example is the same example presented earlier in this paper, where messages' content have the pattern: *content(television,brand(AnyBrand),price(Amount))*; and the *Amount* variable must have a value between 50 and 100.

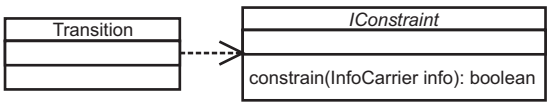

Figure 22: Constraints

```
public class RangeValue implements IConstraint{

  public boolean constrain(InfoCarrier info) {
    String value = (String)info.getValue("Amount");
    if (value==null) return true;
    int intValue = Integer.parseInt(value);
    if (intValue>50 && intValue<100){
      return false; // OK, Should not constrain
    }else{
      return true; // Value out of range, constrain
    }
  }
}
```

Listing 13: Example of Constraint Implementation

3.3.9 Actions

The implementation of actions reuses the trigger module, and at the same time hides from implementors of actions the details of triggers. Then, from the point of view of an implementor of actions, only one interface is available: *IActionFiring*. This interface provides an *execute* method that should be implemented to provide self-healing capabilities.

This transparency and facility of use for implementing actions is achieved through the classes shown in Figure 23. The *Action* class implements *ITrigger* interface, and therefore, it is able to listen for events and execute some behavior once the event occurs. This behavior is represented by the class TriggerFiring, which is extended by the class *ActionFiringAdapter*. The *Action-*

FiringAdapter acts as an Adapter [19] and delegates its behavior to the interface *IActionFiring*. Then, the action behavior is defined by implementing the *IActionFiring* interface.

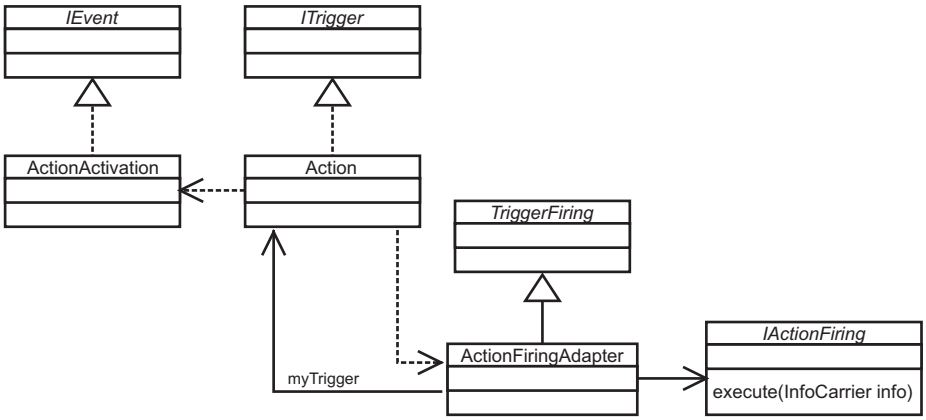

Figure 23: Action Module

3.4 Support for agent developers

Distribution and concurrency distract developers from the application functionalities when developing multi-agent systems. When interaction laws are also present in the system, developers have to design their agents in compliance with the laws, and therefore, the developer's job gets a bit more complex.

The framework do not ignores this problem, and implements a module that support the job done by developers. The module hides agent communication details and provides a set of methods for interact to the law enforcer. Asking for example, which scenes are available. Next sections, provide more details about this module.

3.4.1 Agent

Basically, there are two ways of creating a new agent: building it from scratch or extending the class *Agent* provided by the framework. That class has methods for support the sending and receiving of messages. Moreover, it is fully integrated with the mediator agent and consequently to XMLaw and the interaction model. Even if for some reason, the agent to be developed has to extend another class, it is possible to reuses the Agent class functionality through delegation technique.

Agent class has an instance of the class *ICommunication* as one of its attributes. It represents the channel to where agents send and receive messages. Sending of messages through this channel can be done as follows.

```
this.communication.send(message);
```

However, this communication channel is hidden in the *Agent* class, which provides the methods shown in Listing 14.

```
public void send(Message msg){
    communication.send(msg);
} public Message nextMessage() {
    return communication.nextMessage();
```

```
} public Message waitForMessage() {
    return communication.waitForMessage();
} public Message waitForMessage(long milliseconds) {
    return communication.waitForMessage(milliseconds);
}
```

Listing 14: Sending and Receiving Messages in *Agent* class

3.4.2 Communicating with Mediator: Mediator Protocol

Most of the time agents are unaware of the existence of a mediator. This is because the interception and enforcement of messages is done transparently by the communication module (Section 3.2). However, agents can also communicate with the mediator for asking information about laws specification and execution. Furthermore, mediators may autonomously send messages to agents informing about some law aspects. Those messages are composed of three parts: performative, which specifies the communicative act; content or operation, which specifies either the operation being requested or just some information being transmitted; and a set of parameters, which may be void.

Figure 24 shows the messages that agents can send to the mediator. Such messages request authorization to participate of an organization, request a list of all available organizations under control of the mediator, request some scene instantiation, request scene participation, request a list of all scenes of an organization, and request a list of all scenes that are running.

Mediator agent answers those entire requests through a set of messages shown in Figure 25. Furthermore, mediators can also send messages informing about some situation while enforcing messages. For example, the message *lawException* is sent to an agent when the message sent by the agent to other agent is not allowed according to the specifications.

Performative	Message	Description
request	enterOrganization	Request authorization to enter in an organization
request	listOrganizations	Request a list of all organizations
request	startScene	Request to start a scene execution
request	enterInScene	Request authorization to enter in a scene
request	listScenes	Request a list of all scenes of an organization
request	listRunningScenes	Request a list of all running scenes of a specific scene

Figure 24: Mediator Protocol - Messages that Agents can Send

Performative	Message	Description
inform	sceneAuthorization	Inform a scene authorization
failure	invalidSceneAuthorization	Inform the scene authorization provided is not valid
inform	organizationAuthorization	Inform a organization authorization
failure	invalidOrganizationAuthorization	Inform the organization authorization provided is not valid
failure	organizationDoesNotExist	Organization that agent said does not exist
failure	sceneDoesNotExist	Scene agent said does not exist
inform	organizationList	List of all organizations loaded in the mediator
inform	sceneList	List of all scenes of a certain organization
inform	runningSceneList	List of all running scenes of a specific scene
failure	lawException	Law definitions do not allow the message sent
failure	undefined	Unexpected behavior of the mediator agent

Figure 25: Mediator Protocol - Messages that the Mediator can Send

The Agent class also provides a set of methods to support communication with the mediator agent. Each of these methods sends a message to the mediator, wait for an answer, put the answer in a specific java object and makes this object available to agent's developers. The class Agent is shown in Figure 26.

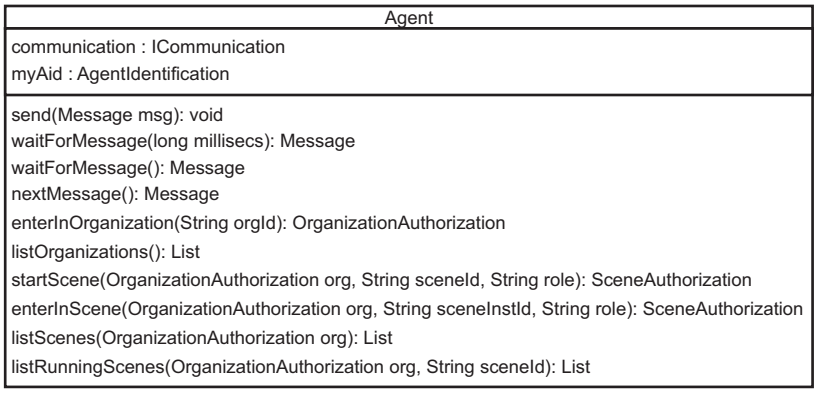

Figure 26: *Agent* Class

4 An example: Trading in a Shopping Center

The example presented in this section is useful to illustrate the using of both the elements from conceptual model and the XMLaw specification. Besides, Section 4.3 shows the framework's modules runtime dynamics when facing the arrival scene.

4.1 The Scenario

Nowadays, large airports do more than just be a place to land and take off airplanes. Shopping centers with hundreds of shops, movie theaters, hotels, business centers, and even gastronomic centers are some of the attractions of modern airports. The large number of potential users and the variety of services offered in a same place turns airports a very promising domain to the development of applications such as flight tickets trading.

The adopted scenario in this example simulates a situation where a person gets at the airport using a mobile device. The airport is equipped with a network of pervasive systems that provides services for movie theaters, dating, searching and trading. When the person (user) gets at the airport, the user requests what services are being provided by the airport. The airport uses a server, named announcer, to send the list of available services to the user. Once the user chooses one of the available services, the announcer informs the user the more specific options related to the selected service. For example, if the user selects the trading service, then the announcer provides the user with a list of the products available to trade. In the example presented in this paper, we focus on the trading service.

Once the user chose the trading service and received a list of all products available to trade, the user selects one of these products and receives a list of all shops that sell the selected product. Then, the user selects one of these shops and starts a negotiation process. If the negotiation succeeds, the user initiates a payment process. In the payment process, one more participant appears: the bank. The bank provides a service that supports the payment of products. In this way,

the user should pay to the bank the product that was negotiated previously. The bank gives the user an electronic receipt that is used to take the product from the shop.

This airport trading system must provide a good level of confidence to their users. It means that users should be protected against malicious behavior of the airport's shops and also the airport's shops should be protected against malicious users. As both parts are interacting using well-defined and public available laws, the confidence in the system as a whole tends to increase.

The next section (Section 4.2) presents how this example uses the elements from the conceptual model and XMLaw to specify the interaction laws.

4.2 Specifying the laws

Interactions between agents that compose the system are modularized using the scene idea. Four scenes are identified: arrival, selection, negotiation and payment. In the first scene, users receives the available services at the airport. In the second one, users select the trading service, receives a list of all products that they can buy, choose one of the products and receives a list of all sellers (shops) for the selected product. In the negotiation scene, users choose one of the sellers and start a negotiation process that may succeed or fail. Finally, in the payment users pay to the bank the dealt product. Figure 27 shows these scenes.

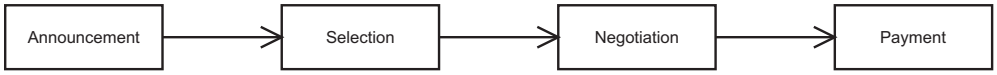

Figure 27: Scenes modularizing interactions

Next, we present the details about each of these scenes. These scenes are represented both through the (informal) graphic notation introduced in Figure 28 and through XMLaw.

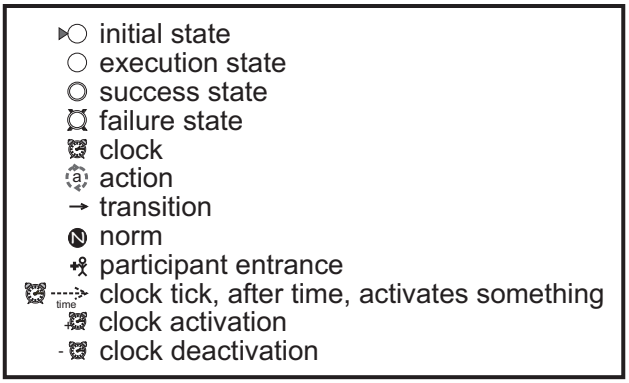

Figure 28: Symbols used to graphically represent scenes

Scene: *arrival*

Each scene specifies the interaction protocol that agents should follow, and consequently what the valid messages are. Figure 29 shows the laws for the arrival scene. The interaction protocol of this scene is illustrated in Figure 30 and it is specified using the XMLaw of Listing 15. It is considered that the whole interaction can not last more than 10 seconds (**time-to-live**). The scene can be created by any agent playing any role (tag *Creator*). However, only agents playing either

customer or *announcer* roles can participate of interactions (tag *Participant*). Agents can only participate of this scene if the protocol's state is in the state *s0* for *customers* participation or *s1* for *announcers* participation. Only two messages are exchanged in this scene. One of them is sent from the customer to the announcer informing that the customer has arrived at the airport (message *m1*). The other one is the announcer's reply to such informing, which contains a list of the available services at the airport (message *m2*). A **clock** is activated when an customer agent sends a message requesting the available services. The goal of this clock is to verify if the systems is achieving a good reply-time. Then, if the announcer does not reply within this time, an action (*announcer-is-down*) is activated and a new announcer is turned on.

Scene: Arrival
1. It is allowed that every agent creates this scene.
2. Agents playing the "customer" role can only enter in the scene if the protocol is in the initial state, that is, no conversation has occured.
3. Agents playing the "announcer" role can only enter in the scene after an agent "customer" has started the conversation.
4. In order to start a conversation, the "customer" agent should send a message introducing itself to the agent "announcer", that in its turn replies with a list of available services.
5. When the "customer" agent starts the conversation, the "announcer" agent has 5 seconds to send a reply. In case of no reply after the 5 seconds, it could mean that, for example, the "announcer" agent is not working properly or there are some network communication problems. In this case, an action should be executed in order to try to recover from the failure.

Figure 29: Arrival Scene's Laws

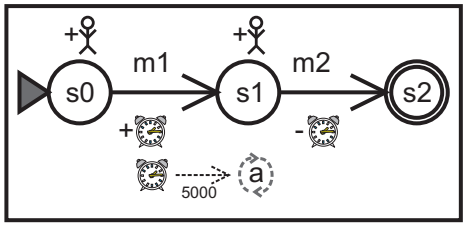

Figure 30: Scene: arrival

```
<Scene id="anouncement" time-to-live="10000"> <!-- 10 sec. -->
    <Creators>
        <Creator agent="any" role="any"/>
    </Creators>
    <Entrance>
        <Participant agent="any" role="customer">
            <StatesRef>
                <StateRef ref="s0"/>
            </StatesRef>
        </Participant>
        <Participant agent="any" role="announcer">
            <StatesRef>
                <StateRef ref="s1"/>
            </StatesRef>
        </Participant>
    </Entrance>
    <Messages>
        <Message id="m1" template="message(request,sender(_,customer),receiver(_,announcer),content(hello))."/>
        <Message id="m2" template="message(inform,sender(_,announcer),receiver(CustomerAgent,customer),content(
            services([_|_]))."/>
    </Messages>
    <Protocol>
        <States>
            <State id="s0" type="initial" label="Initial state"/>
            <State id="s1" type="execution" label="Message sent"/>
            <State id="s2" type="success" label="Response answered"/>
        </States>
        <Transitions>
            <Transition id="t1" from="s0" to="s1" message-ref="m1"/>
            <Transition id="t2" from="s1" to="s2" message-ref="m2"/>
        </Transitions>
    </Protocol>
    <Clocks>
        <Clock id="time-for-answering-hello" type="regular" tick-period="5000"> <!-- 5 sec. -->
            <Activations>
                <Element ref="t1" event-type="transition_activation"/>
            </Activations>
            <Deactivations>
                <Element ref="t2" event-type="transition_activation"/>
            </Deactivations>
        </Clock>
    </Clocks>
    <Actions>
        <Action id="announcer-is-down" class="br.pucrio.inf.les.law.app.airport.repairactions.HealAnnouncerAction">
            <Element ref="time-for-answering-hello" event-type="clock_tick"/>
        </Action>
    </Actions>
</Scene>
```

Listing 15: Arrival Scene: XMLaw

Scene: *selection*

This scene may last up to 5 minutes, which approximates the amount of time given to the user (*customer* agent) to choose among the services and products. The first message is sent from the customer to the announcer informing the selected service. This message causes the firing of transition *t3* (in the XMLaw). The announcer's reply contains the list of available products for the requested service[5] (message *m4*). The customer chooses one of the products and requests the list of sellers (message *m5*). Finally, the list of sellers is informed through message *m6* and the protocol reaches the success state *s7*. Then, both protocol and scene finish.

```
<Scene id="selection" time-to-live="300000"> <!-- 5 min. -->
    <Creators>
        <Creator agent="any" role="any"/>
    </Creators>
    <Entrance>
        <Participant agent="any" role="customer">
            <StatesRef>
                <StateRef ref="s3"/>
            </States>
        </Participant>
        <Participant agent="any" role="announcer">
            <StatesRef>
                <StateRef ref="s4"/>
            </StatesRef>
```

[5] In this case study only the trading service is considered.

Scene: selection
1. It is allowed that every agent creates this scene.
2. “Customer” agents are in charge of starting the conversation and, therefore, they can only enter in the scene if the protocol is in the initial state.
3. An “announcer” agent is only allowed to enter in this scene when replying a previous message sent by a “customer” agent.

Figure 31: Selection Scene’s Laws

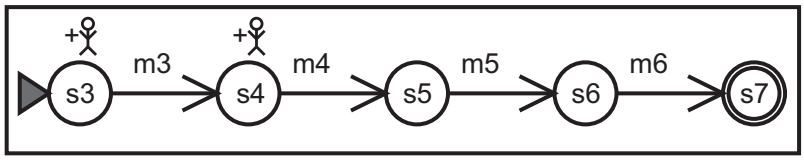

Figure 32: Scene: selection

```
        </Participant>
    </Entrance>
    <Messages>
        <Message id="m3" template="message(request,sender(_,customer),receiver(_,announcer),content(option(Service))).
            "/>
        <Message id="m4" template="message(inform,sender(_,announcer),receiver(_,customer),content(products(
            ListOfProducts))).">/>
        <Message id="m5" template="message(request,sender(_,customer),receiver(_,announcer),content(sellers-of(Product
            ))).">/>
        <Message id="m6" template="message(inform,sender(_,announcer),receiver(Customer,customer),content(sellers([_|_
            ]))).">/>
    </Messages>
    <Protocol>
        <States>
            <State id="s3" type="initial" label="Ready to ask for options"/>
            <State id="s4" type="execution" label="List of products requested"/>
            <State id="s5" type="execution" label="List of products informed"/>
            <State id="s6" type="execution" label="List of sellers requested"/>
            <State id="s7" type="success" label="List of sellers informed"/>
        </States>
        <Transitions>
            <Transition id="t3" from="s3" to="s4" message-ref="m3"/>
            <Transition id="t4" from="s4" to="s5" message-ref="m4"/>
            <Transition id="t5" from="s5" to="s6" message-ref="m5"/>
            <Transition id="t6" from="s6" to="s7" message-ref="m6"/>
        </Transitions>
    </Protocol>
</Scene>
```

Listing 16: Selection Scene: XMLaw

Scene: *negotiation*

The interaction protocol used in this scene is based on FIPA-CONTRACT-NET [20]. The first message is sent from a costumer to a seller requesting a proposal for the selected product. This message contains some information about the maximum price customer can pay for the product and about the preferred brand. Sellers can only send proposals with value less or equal the maximum price specified by the customer. This price constraint is implemented through the class *EnforceValue*, showed in Listing 18. This class is called by the constraints in the transitions *t7* and *t9*. In *t7* the maximum price is obtained and in *t9* the informed value is verified. Sellers can reply a customer request through a proposal (message *m8* and transition *s9* to *s10*) or refusing to send a proposal (message *m11*). If the seller refuses, then the protocol finishes in the *s13* failure state. Otherwise, when the customer agent receives a proposal, it has 20 seconds to decide whether

accepting or rejecting the proposal. If after the 20 seconds the customer has not decided yet, the seller receives a permission to cancel the proposal previously made. This law is important because it protects the seller in the case where another client is interested in the same product but the seller cannot sell because it has already started a negotiation for this product. The state *s15* represents when a seller has cancelled a proposal after the 20 seconds of "customer indecision". The state *s14* means that the customer has not accepted the proposal made by a seller. Finally, if the customer has accepted the proposal, the protocol goes to state *s11* and the seller informs where the payment should be done (message *m10*).

Scene: negotiation
1. It is allowed that every agent creates this scene.
2. “Customer” agents are in charge of starting the conversation and, therefore, they can only enter in the scene if the protocol is in the initial state.
3. A “seller” agent is only allowed to enter in this scene when replying a previous message sent by a “customer” agent.
4. “Customer” agents have 20 seconds to decide whether they accept or reject a proposal sent by a “seller”. When the 20 seconds has elapsed, the “seller” obtains a permission to cancel the negotiations with the “customer” agent.
5. Agents that play the “seller” role must send proposals with the price always less or equals to the suggested price by the “customer”.

Figure 33: Negotiation Scene’s Laws

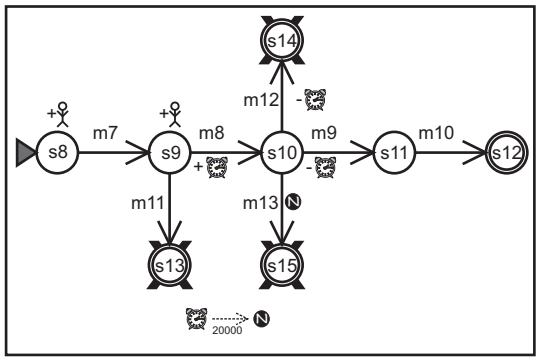

Figure 34: Scene: negotiation

```
<Scene id="negotiation" time-to-live="infinity">
    <Creators>
        <Creator agent="any" role="any"/>
    </Creators>
    <Entrance>
        <Participant agent="any" role="customer">
            <StatesRef>
                <StateRef ref="s8"/>
            </StatesRef>
        </Participant>
        <Participant agent="any" role="seller">
            <StatesRef>
                <StateRef ref="s9"/>
            </StatesRef>
        </Participant>
    </Entrance>
    <Messages>
        <Message id="m7" template="message(cfp,sender(_,customer),receiver(_,seller),content(product-details(product(
            Product),maxprice(Price),brand(Brand)))."/>
        <Message id="m8" template="message(propose,sender(SellerAgent,seller),receiver(_,customer),content(product(
            name(Product),price(Price),brand(Brand)))."/>
        <Message id="m9" template="message(accept-proposal,sender(_,customer),receiver(_,seller),content(_))."/>
        <Message id="m10" template="message(inform,sender(_,seller),receiver(CustomerAgent,customer),content(payto(
            Bank)))."/>
        <Message id="m11" template="message(refuse,sender(_,seller),receiver(_,customer),content(Reason))."/>
        <Message id="m12" template="message(reject-proposal,sender(_,customer),receiver(_,seller),content(Reason))."/>
        <Message id="m13" template="message(cancel,sender(_,seller),receiver(_,customer),content(Reason))."/>
    </Messages>
    <Protocol>
        <States>
            <State id="s8"  type="initial" label="Ready␣for␣starting␣negotiations"/>
            <State id="s9"  type="execution" label="Call␣for␣proposal␣requested"/>
            <State id="s10" type="execution" label="Proposal␣sent"/>
            <State id="s11" type="execution" label="Proposal␣accepted"/>
            <State id="s12" type="success" label="Bank␣informed"/>
            <State id="s13" type="failure" label="Refuse␣sending␣proposal"/>
            <State id="s14" type="failure" label="Proposal␣rejected"/>
            <State id="s15" type="failure" label="Too␣long␣time␣to␣decide"/>
        </States>
        <Transitions>
            <Transition id="t7" from="s8" to="s9" message-ref="m7">
                <Constraints>
                    <Constraint class="br.pucrio.inf.les.law.app.airport.repairactions.EnforceValue">
                        <Semantic>Gets value on price</Semantic>
                    </Constraint>
                </Constraints>
            </Transition>
            <Transition id="t8" from="s9" to="s10" message-ref="m8">
                <Constraints>
                    <Constraint class="br.pucrio.inf.les.law.app.airport.repairactions.EnforceValue">
                        <Semantic>Enforces value on price</Semantic>
                    </Constraint>
                </Constraints>
            </Transition>
            <Transition id="t9" from="s10" to="s11" message-ref="m9"/>
            <Transition id="t10" from="s11" to="s12" message-ref="m10"/>
            <Transition id="t11" from="s9" to="s13" message-ref="m11"/>
            <Transition id="t12" from="s10" to="s14" message-ref="m12"/>
            <Transition id="t13" from="s10" to="s15" message-ref="m13">
                <ActiveNorms>
                    <Norm ref="seller-permission-to-cancel"/>
                </ActiveNorms>
            </Transition>
        </Transitions>
    </Protocol>
    <Clocks>
        <Clock id="time-to-decide" type="regular" tick-period="20000"> <!-- 20 sec. -->
            <Activations>
                <Element ref="t8" event-type="transition_activation"/>
            </Activations>
            <Deactivations>
                <Element ref="t9" event-type="transition_activation"/>
                <Element ref="t12" event-type="transition_activation"/>
            </Deactivations>
        </Clock>
    </Clocks>
    <Norms>
        <Permission id="seller-permission-to-cancel">
            <Owner>SellerAgent</Owner>
            <Activations>
                <Element ref="time-to-decide" event-type="clock_tick"/>
            </Activations>
        </Permission>
    </Norms>
</Scene>
```

Listing 17: Negotiation Scene: XMLaw

```
public class EnforceValue implements IConstraint{

    private double maxPrice;
    private double price;

    public boolean constrain(InfoCarrier info) {

        Message msg = (Message)info.getValue(Constants.MESSAGE_INFO_KEY);
        if ( msg.getPerformative().equals(Message.CALL_FOR_PROPOSAL) ){
            maxPrice = Double.parseDouble((String)info.getValue("Price"));
        }else if ( msg.getPerformative().equals(Message.PROPOSE) ){
            price = Double.parseDouble((String)info.getValue("Price"));
            if (price <= maxPrice){
                return false;
            }else{
                return true;
            }
        }
        return false;
    }
}
```

Listing 18: Contraining the price

Scene: *payment*

The creation of a payment scene can be only be performed by agents playing the customer role and that have the permission *permission-to-pay*. This permission is obtained when a customer agent successfully completes a negotiation scene. This permission is specified in the context of the organization, and therefore it is global. Listing 19 shows the specification of this permission.

```
    ...
    </Scenes>
    <Norms>
        <Permission id="permission-to-pay">
            <Owner>CustomerAgent</Owner>
            <Activations>
                <Element ref="negotiation" event-type="sucessful_scene_completion"/>
            </Activations>
            <Deactivations>
                <Element ref="payment" event-type="sucessful_scene_completion"/>
            </Deactivations>
        </Permission>
    </Norms>
</LawOrganization>
```

Listing 19: Permission specification: limiting the access to payment scene

Except for the access control introduced through the permission *permission-to-pay*, this scene is very simple and its specification is presented in Figure 36 and Listing 20.

Scene: payment
1. Only “customer” agents that have successful completed the negotiation scene can enter in this payment scene. 2. “Customer” agents can only enter in this scene if the protocol is in the initial state. 3. A “bank” agent is only allowed to enter in this scene when replying a previous message sent by a “customer” agent.

Figure 35: Payment Scene’s Laws

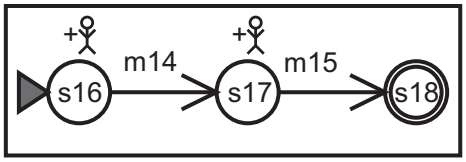

Figure 36: Scene: *payment*

```
<Scene id="payment" time-to-live="infinity">
    <Creators>
        <Creator agent="any" role="customer">
            <ActiveNorms>
                <Norm ref="permission-to-pay"/>
            </ActiveNorms>
        </Creator>
    </Creators>
    <Entrance>
        <Participant agent="any" role="customer">
            <StatesRef>
                <StateRef ref="s16"/>
            </StatesRef>
        </Participant>
        <Participant agent="any" role="bank">
            <StatesRef>
                <StateRef ref="s17"/>
            </StatesRef>
        </Participant>
    </Entrance>
    <Messages>
        <Message id="ml4" template="message(request,sender(_,initiator),receiver(_,participant),content(pay(amount(
            Amount),to(Seller)))).">
        <Message id="ml5" template="message(inform,sender(_,participant),receiver(_,initiator),content(receipt(number(
            Number)))).">
    </Messages>
    <Protocol>
        <States>
            <State id="s16" type="initial" label="Ready␣for␣payment"/>
            <State id="s17" type="execution" label="Payment␣order␣emitted"/>
            <State id="s18" type="success" label="Receipt␣sent"/>
        </States>
        <Transitions>
            <Transition id="t14" from="s16" to="s17" message-ref="ml4"/>
            <Transition id="t15" from="s17" to="s18" message-ref="ml5"/>
        </Transitions>
    </Protocol>
</Scene>
```

Listing 20: Payment Scene: XMLaw

4.3 Modules Dynamics

This section shows how the main modules of the framework behave in face of an interaction in the context of the arrival scene. It is described in a sequence of steps showing the consequence of the agents' interactions and the law specification.

1. Customer agent sends the message through the Agent class' method sendMessage.
   message(request, sender(maria@JADE,customer),receiver(jose@JADE,announcer),
   content(hello)).

2. Message gets at the agent communication module, which redirects the message to the mediator agent.

3. The mediator agent receives the message from its communication module, identifies to which scene the message is addressed and fires a message arrival event in the context of the scene.

4. The protocol of the scene class is an observer of message arrival events that occurs in the context of the scene. Hence, the protocol is notified about the message arrival.

5. It verifies that the current state is "s0" and there is a transition from "s0" to "s1" that is should fire when a message matches the template of message "m1". The charging of performing this matching is delegated to the PatternMatcher class. Then, as the pattern matches, the transition "t1" is fired and a transition activation event is generated in the context of the scene.

6. The clock "time-for-answering-hello" is waiting for the occurrence of the activation of transition "t1" to be activated. As this event just happened, this clock is then activated is starts to count the time. In other words, a Clock object generates an instance of ClockActivation class, which represents an event, and after that, the clock object also creates an instance of ActivatedClock that starts to count the time.

7. Supposing that after 5 seconds the agent (jose@JADE,announcer) has not replied the previous request message, the ActivatedClock generates a ClockTick instance, which is an event.

8. Then to finalize, the action "announcer-is-down" is waiting for the occurrence of clock ticks events, which also just happened. This action is then activated, and a new instance of the class specified in the XMLaw is created, and its execute method is invoked. In this case, the action is implemented by the HealAnnouncerAction class.

5 Related Work

Related work regarding this paper is done in two ways. The first one relates the implementation of other law approaches with the one presented here. The second one could compare the model used to specify interaction with other existent models.

Following the first option, the work presented in this paper implements a law enforcement approach as an object-oriented framework, which brings the benefits of reuse and flexibility. Moreover, to the best of our knowledge, other law-based approaches have not published their implementations, which make tough comparing with this one. In this sense, we believe that the design strategy presented throughout this paper is a novel and important contribution to the field of law-based approaches.

Regarding the second approach, it is possible to cite at least three important researches with goals very similar to the conceptual model used here. In Esteva [11] approach, scenes are similar to the protocol elements proposed in this paper. Both Esteva scenes and protocol elements specify the interaction protocol using a global view of the interaction. It means that all the interaction among the agents is specified in only one protocol in opposite to individual agent views, where many partial views of the protocol (one of each agent) are specified. The time aspect is represented in Esteva approach as timeouts. Timeouts allow activating transitions after a given number of time units passed since a state was reached. On the other hand, due to our event model, the clock element proposed in this paper can both activate and deactivate not only transitions, but also other clocks and norms. Connecting clocks to norms allows a more expressive normative behavior; norms become time sensitive elements. Furthermore, we also include the concept of actions, which allows execution of java code in response to some interaction situation.

OMNI [21] is a framework for modeling agent organizations. This framework is composed of three dimensions: normative, organizational and ontological. These dimensions aim to cover

from analysis to implementation of agent organizations. In the normative dimension, developers specify the mechanisms of social order, in terms of common norms and rules, which members are expected to adhere to. The organizational dimension describes the structure of an organization. The ontological dimension defines environment and contextual relations and communication aspects in organizations. In addition, each one of these dimensions can be viewed in three abstraction levels: abstract, concrete and implementation. In the abstract level, the general organization goals are defined in a high level of abstraction. It also contains the definition of the ontology of the model itself. Based on the abstract level, norms, rules, roles, interaction protocols and concrete ontological concepts are defined. The implementation level assumes a given multi-agent architecture as basis for the implementation of the organizational model, and also mechanisms for role enactment and norm enforcement.

Minsky [22, 23] proposes a coordination and control mechanism called law governed interaction (LGI). This mechanism is based in two basic principles: the local nature of the LGI laws and a decentralization of law enforcement. The local nature of LGI laws means that a law can regulate explicitly only local events at individual home agents, where home agent is the agent being regulated by the laws; the ruling for an event e can depend only on e itself, and on the local home agent's context; and the ruling for an event can mandate only local operations to be carried out at the home agent. On the other hand, the decentralization of law enforcement is an architectural decision argued as necessary for achieving scalability. Furthermore, it provides a language to specify laws and it is concerned with architectural decisions to achieve a high degree of robustness. In contrast, our approach provides a explicit conceptual model and focuses on different concepts such as Scenes, Norms and Clocks.

6 Conclusions

This paper presented an object-oriented framework that supports a model for specifying agent's interaction. The framework monitors agent's interactions and verify if they are in accordance with the XMLaw specification. The framework is flexible once it provides hotspots to plug in different communication mechanisms and different content languages (prolog, DAML+OIL). Moreover, the report given in this paper provides a guide in the sense that new agent enforcement frameworks can appear based on ideas presented here.

We believe that development of open multi-agent systems requires new techniques and abstractions to deal with the many sources of complexity that arises in the scenario, and the proposed framework tackles a very specific and important point to achieve higher levels of system's confidence: taming agent's autonomy.

However, we believe that more sophisticated mechanisms should be incorporated in the framework, such as reputation mechanisms, fault-tolerance techniques, and a more scalable strategy to mediate the messages (as opposed to the centralized mediator presented in this framework).

Two current research trends are being done. One is related on how to incorporate in the agents the ability to reason about the laws in order to plan their actions. The other trend is related to improving the conceptual model to allow composition of elements. For example, we could specify a generic scene and extend this scene to a more specific context.

References

[1] M. Wooldridge, G. Weiß, P. Ciancarini (Eds.), Agent-Oriented Software Engineering II, Second International Workshop, AOSE 2001, Montreal, Canada, May 29, 2001, Revised Papers and Invited Contributions, Vol. 2222 of Lecture Notes in Computer Science, Springer, 2002.

[2] N. Sadeh, R. Arunachalam, J. Eriksson, N. Finne, S. Janson, Tac-03: a supply-chain trading competition, AI Mag. 24 (1) (2003) 92–94.

[3] F. Zambonelli, N. Jennings, M. Wooldridge, Developing multiagent systems: The gaia methodology, ACM Trans. Softw. Eng. Methodol. 12 (3) (2003) 317–370.

[4] R. Paes, G. Carvalho, C. Lucena, P. Alencar, H. Almeida, V. T. da Silva, Specifying laws in open multi-agent systems, in: Agents, Norms and Institutions for Regulated Multiagent Systems - ANIREM, Utrecht, The Netherlands, 2005.

[5] J. F. Hubner, J. S. Sichman, O. Boissier, A model for the structural, functional, and deontic specification of organizations in multiagent systems, in: XVI Brazilian Symposium on Artificial Intelligence, Porto de Galinhas, Brazil, 2002.

[6] Z. Guessoum, N. Faci, J.-P. Briot, Adaptive replication of large-scale multi-agent systems - towards a fault-tolerant multi-agent platform, in: Software Engineering for Large-Scale Multi-Agent Systems - SELMAS'05, St. Louis, Missouri - USA, 2005.

[7] WOSS '02: Proceedings of the first workshop on Self-healing systems, ACM Press, 2002.

[8] J. O. Kephart, D. M. Chess, The vision of autonomic computing, IEEE Computer Magazine 36 (1) (2003) 41–50.

[9] M. Fayad, D. C. Schmidt, Object-oriented application frameworks, Communications of ACM 40 (10) (1997) 32–38.

[10] T. Murata, N. H. Minsky, On monitoring and steering in large-scale multi-agent systems, in: Selmas'03 2nd International Workshop on Software Engineering for Large-Scale Multi-Agent Systems, Portland, Oregon, 2003.

[11] M. Esteva, Electronic institutions: from specification to development, Ph.D. thesis, Institut d'Investigació en Intel.ligència Artificial, Catalonia - Spain (October 2003).

[12] F. J. Martín, E. Plaza, J. A. Rodriguez-Aguilar, An infrastructure for agent-based systems: an interagent approach, International Journal of Intelligent Systems (IJIS) (1999) 217–240.

[13] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, D. Winer, Simple object access protocol (soap) 1.1 (May 2000).
URL `http://www.w3.org/TR/SOAP/`

[14] Fipa, Foundation for intelligent physical agents (2003).
URL `http://www.fipa.org/`

[15] F. Bellifemine, A. Poggi, G. Rimassa, Jade: a fipa2000 compliant agent development environment, in: Proceedings of the fifth international conference on Autonomous agents, ACM Press, 2001, pp. 216–217.

[16] G. Cugola, E. D. Nitto, A. Fuggetta, Exploiting an event-based infrastructure to develop complex distributed systems, in: Proceedings of the 20th international conference on Software engineering, IEEE Computer Society, 1998, pp. 261–270.

[17] J. Widom, S. Ceri, Active Database Systems: Triggers and Rules For Advanced Database Processing., Morgan Kaufmann, 1996.

[18] P. B. Menezes, Linguagens Formais e Autômatos, Série Livros Didáticos, Sagra Luzzato Editores, Porto Alegre, 1997.

[19] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: elements of reusable object-oriented software, Addison-Wesley, 1995.

[20] F. F. for Intelligent Physical Agents, Fipa contract net interaction protocol specification (December 2002).
URL http://www.fipa.org/ specs/ fipa00029/

[21] J. Vázquez-Salceda, V. Dignum, F. Dignum, Organizing multiagent systems, Tech. rep., Institute of Information & Computing Sciences (March 2004).

[22] N. H. Minsky, T. Murata, On manageability and robustness of open multi-agent systems, in: C. Lucena, A. Garcia, A. Romanovsky, J. Castro, P. Alencar (Eds.), Software Engineering for Multi-Agent Systems II: Research Issues and Practical Application, Vol. 2940/2004, Springer-Verlag Heidelberg, 2004, pp. 189–206.

[23] N. H. Minsky, V. Ungureanu, Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems, ACM Trans. Softw. Eng. Methodol. 9 (3) (2000) 273–305.