Domain Engineering to Ensure Flexibility on Interaction Laws of Multi-Agent Systems

Authors: Gustavo Carvalho[1], Rodrigo Paes[1], Carlos Lucena[1], Ricardo Choren[2]

Affiliations:

1 DI – PUC-Rio, Rua M. de São Vicente 225, Gávea, 22453-900, Rio de Janeiro/RJ, Brazil

2 SE/8 – IME, Pça Gen Tibúrcio 80, Praia Vermelha, 22290-270, Rio de Janeiro/RJ, Brazil

Correspondent author: Gustavo Carvalho

Fax: 21 2511-5645

e-mail: guga@les.inf.puc-rio.br

Keywords:

Open multi-agent systems, maintainability, reuse, law enforcement, interaction protocol

Abstract:

Law enforcement approaches have been proposed to promote dependability in open multi-agent systems. Interaction laws are defined and then enforced to promote predictability. As new software demands and requirements appear, the system and its interaction laws must evolve to support those changes. Poor support for evolution has a negative impact on system maintainability. The purpose of domain engineering (DE) is to produce a set of reusable assets for a family of systems, which are then used to build concrete members of the family. DE represents the process of development for reuse. Flexibility is the ease with which a system or component can be modified for use in applications other than those for which it was originally designed. Languages and models should support flexibility to minimize the costs of these changes. In this paper, we discuss how we have designed the M-Law infrastructure to support interaction law evolution providing support to produce a set of reusable laws for a family of systems. As an example of this idea, we have implemented two customizable applications in the area of electronic negotiation expressed as an open system environment.

I. INTRODUCTION

Usually, distributed software agents are independently implemented, i.e., the development is done without a centralized control. Nevertheless, every agent developer should have a priori access to the open multi-agent systems (MAS) specification. Open MAS agents are autonomous, i.e. they may behave unpredictably, which can lead the system to an undesirable state. Thus the specification of open MAS should include protocol descriptions and interaction laws that define what and when something can happen in an open system. Laws are restrictions imposed by the environment to tame uncertainty and to promote open system dependability [15, 16]. A governance mechanism is the mediator that enforces the law specification. Examples of governance mechanisms are LGI [15], Islander [8] and MLaw [17].

Sometimes, the interaction laws that define the relationships between agents are not always fully understood early in the open MAS life cycle. Moreover, many more interaction laws are not applied because of a lack of system support for changing interaction laws (i.e. extensibility) or because the interaction laws are exceptionally complex. One of the most important attributes of system dependability consists of the ability to undergo repairs and modifications, i.e., maintainability. In this paper, software maintainability is defined as the ease of extending software to fulfill specific domain requirements, and we focus on design time support for interaction law maintainability for open MAS.

Thus open MAS specification should also be developed to facilitate extensions and law-governed approaches should also present a solution to this issue. As open MAS must be customized according to different purposes and peculiarities, extensions over agent interactions can be expressed. In this paper, we introduce a design support to facilitate the changes of laws in MLaw [16]. During the interpretation of laws, the description is mapped to an execution model in two steps; the first step provides the abstract execution model, with

some hooks that will be fulfilled with law extensions. We also present how we enhanced the XMLaw description language with some refinement operators [6] to specify extensible laws. These refinement operators are used to map extensible (customizable) law specifications to the MLaw governance mechanism monitor thus providing the means for seamless law maintainability for open MAS. The main contribution of this paper is to illustrate how the refinement operators are mapped to MLaw structure.

This paper is organized as follows. Section 2 details the law-governed approach, its architecture (MLaw) and some elements of the XMLaw description language [16]. In Section 3, we discuss variations in open MAS interactions and we describe how we included refinement operators in XMLaw. We also discuss the design of MLaw that was proposed to facilitate law extensions. Section 4 details an experiment based on a real world application. Section 5 shows a sample application with some scenarios that include extension points identified in a negotiation application and we show how XMLaw can be used to support extensibility in a compliance mechanism. Related work is described in Section 6. Finally, we evaluate this approach and describe some future work and our conclusions in Section 7.

II. GOVERNING INTERACTIONS IN OPEN MAS WITH XMLaw

Law-governed architectures are designed to guarantee that the specifications of open systems will be obeyed. The core of a law-governed approach is the mechanism used by the mediators to monitor the conversations between components. We have developed M-Law [17], an architecture that provides a communication component, or mediator, for enforcing interaction laws. M-Law was designed to allow extensibility in order to fulfill open system requirements or interoperability concerns.

M-Law was built to support law specification using XMLaw [16]. XMLaw is used to represent the interaction rules of an open system specification. These rules are interpreted by the M-Law mediator that, at runtime, analyzes the compliance of agents with interaction laws

specifications. A law specification is a description of law elements which are interrelated in a way that it is possible to specify interaction protocols using time restrictions, norms, or even time sensitive norms. XMLaw follows an event-driven approach, i.e., law elements communicate by the exchange of events.

The XMLaw conceptual model (Figure 1) uses the abstraction of Scenes to help to organize interactions. The idea of scenes is similar to theater plays, where actors play according to well defined scripts, and the whole play is composed of many scenes sequentially connected. Scenes are composed of Protocols, Constraints, Clocks, and Norms. It means that these four elements share a common interaction context through the scenes. Since protocols define the interaction among the agents, different protocols should be specified in different scenes.

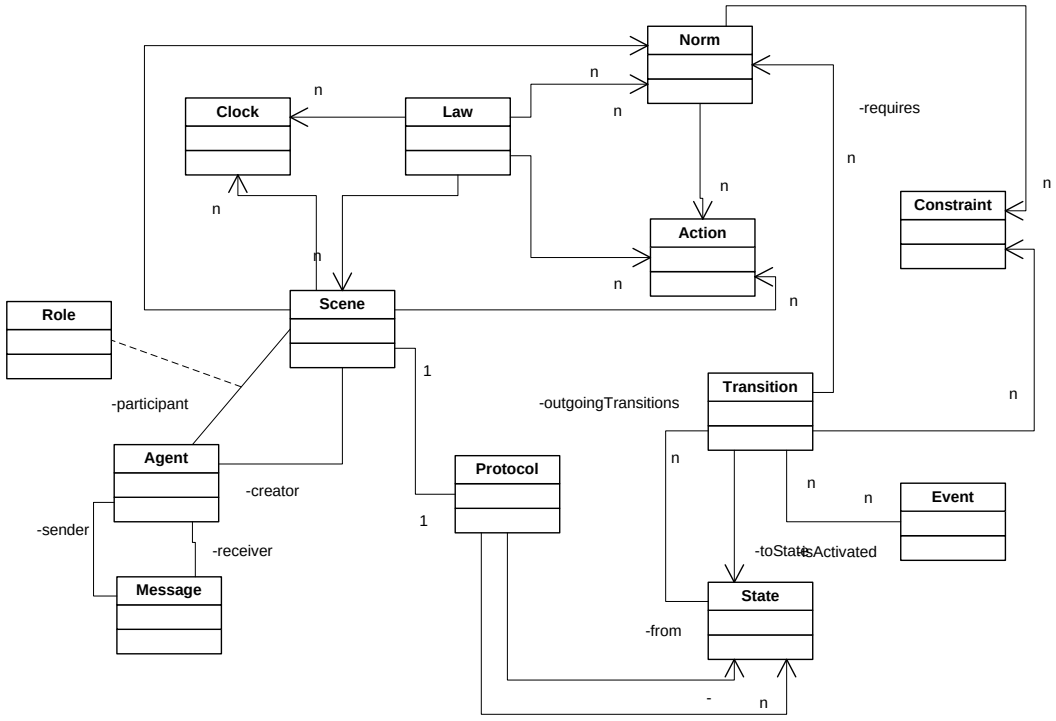

Figure 1. XMLaw conceptual metamodel.

Statically, an interaction protocol defines the set of states and transitions (activated by messages or any other kind of event) allowed for agents in an open system. Norms are jointly used with the protocol specification, constraints, actions and also temporal elements, to

provide a dynamic configuration for the allowed behavior of agents in an open system. The mediator keeps information about all data regarding system execution such as the set of activated and deactivated enforcement elements.

Laws may be time sensitive, e.g., although an element may be active at time t1, it may not at time t2 (t1 < t2). XMLaw provides the Clock element to take care of the timing aspect. Clocks, used to indicate that a certain period has elapsed, are activated and deactivated by law elements and, once active, they produce clock-tick events. In other words, a clock represents time restrictions or controls and they can be used to activate other law elements.

II.1 NORMS

A Norm is an element used to enable or disable agents' conversation paths. For instance, a norm can forbid an agent to interact in a negotiation scene. There are three types of norms with different semantics in XMLaw: obligations, permissions and prohibitions. The obligation norm defines a commitment that software agents acquire while interacting with other entities. For instance, the winner of an auction is obligated to pay the committed value and this commitment might contain some penalties to avoid breaking this rule. The permission norm defines the rights of a software agent at a given moment, e.g. the winner of an auction has permission to interact with a bank provider through a payment protocol. Finally, the prohibition norm defines forbidden actions of a software agent at a given moment; for instance, if an agent does not pay its debts, it will not be allowed future participation in a scene.

The structure of the Permission (Listing 1), Obligation and Prohibition elements are equal. Each type of norm contains activation and deactivation conditions. In Listing 1, an assembler will receive the permission upon logging in to the scene (scene activation event) and will lose the permission after issuing an order (event orderTransition). Furthermore, norms define the agent role that owns it through the attribute Owner. In Listing 1, the

assembler agent will receive the permission. Norms also have constraints and actions associated with them, but these elements will be explained later. Norms also generate activation and deactivation events. For instance, as a consequence of the relationship between norms and transitions, it is possible to specify which norms must be made active or deactivated for firing a transition. In this sense, a transition could only fire if the sender agent has a specific norm.

```
<Norm type="permission" id="AssemblerPermissionRFQ">
   <Assignee role-ref="assembler" role-instance="$assembler.instance"/>
   <Activations>
      <Element ref="negotiation" event-type="scene_creation"/></Activations>
   <Deactivations>
     <Element ref="orderTransition" event-type="transition_activation"/></Deactivations>
   <Constraints>
      <Constraint id="checkCounter" class="CounterLimit"/></Constraints>
   <Actions>
      <Action id="permissionRenew" class="ZeroCounter">
         <Element ref="nextDay" event-type="clock_tick"/></Action>
      <Action id="orderID" class="RFQCounter"/>
        <Element ref="rfqTransition" event-type="transition_activation"/></Action>
    </Actions>
</Norm>
```

Listing 1. Permission structure

II.2 CONSTRAINTS

A constraint is a restriction over norms or transitions and, generally, it specifies filters for events, constraining the allowed values for a specific attribute of an event. For instance, messages carry information that is enforced in various ways. A message pattern enforces the message structure fields [16]. A message pattern does not describe what the allowed values for specific attributes are, but constraints can be used for this purpose. In this way, developers are free to build as complex constraints as needed for their applications.

Constraints are defined inside the Transition (Listing 2) or Norm (Listing 1) elements. Constraints are implemented using Java code. The Constraint element defines the class attribute that indicates the Java class that implements the filter. This class is called when a transition or a norm is supposed to fire, and basically the constraint analyzes if the message values or any other events' attributes are valid. Listing 2 shows a constraint that verifies if the date expressed in a message is valid; if it is not, the message will be blocked. In Listing 1, a

constraint is used to verify the number of messages that the agent has sent until now; if it has been exceeded, the permission is no longer valid.

```
<Transition id="rfqTransition" from="as1" to="as2" message-ref="rfq>
   <Constraints>
      <Constraint id="checkDueDate" class="ValidDate"/></Constraints>...
</Transition>
```

Listing 2. A Constraint in a Transition tag.

II.3 ACTIONS

An action is a domain-specific Java code that runs integrated with XMLaw specifications. Actions can be used to plug services in the mediator. For instance, the mediator can call a debit service from a bank agent to automatically charge the purchase of an item during a negotiation. In XMLaw, an action can be defined in three different scopes: Organization, Scene and Norms.

Since actions are also XMLaw elements, they can be activated by any event such as transition activation, norm activation and even action activation. The action structure is showed in the example of Listing 1 (in this example, a norm action). The class attribute of an Action specifies the Java class in charge of the functionality implementation. The Element tag references the events that activate this action, and as many Element tags as needed can be defined to trigger an action.

III. REFINEMENT OPERATORS TO SPECIFY LAWS IN OPEN MAS

Interaction laws in open MAS should be specified and developed to facilitate maintainability and evolution. Thus, it is necessary to specify which law elements can be customized and so defined as extension points. We propose to define extension points as a means of representing knowledge about the place where modifications and enhancements in laws can be made. In our context, it is useful to permit the inclusion of norms, constraints, actions or any other law elements into a pre-defined law specification. Even with extension points, the semi-complete law element specification can be referenced by other law elements. The subsections below explain how the interaction specification with extension points can be

better prepared for further refinements. We explain how the abstract and extends operators in XMLaw specification are implemented in M-Law. Further, we detail the design to support the implementation of this feature. We also describe the impact of this feature in the mediator lifecycle.

III.1 THE ABSTRACT OPERATOR

The clear documentation of an extension point is very important. At implementation level, the refinement operators must help the designer to find out where the semi-complete specifications are. The *abstract* operator defines when a law element is not completely implemented. For instance, it can be used to defer the definition of the implementation of actions and constraints classes and also to extend the definition of other law elements.

The *abstract* operator is used to indicate in XMLaw code when we have “hooks” or even when the existing laws must be better defined to be used. If no value is determined, the element is a concrete one (default abstract=”false”). If the designer wants to specify that a law element needs some refinements to be used, he has to explicitly specify the attribute abstract with the value true (abstract=”true”). An example of extension is given in the specification of the relationship between orders and offers in a negotiation protocol. Suppose the following scenario based on [7]: agents confirm supplier offers by issuing orders. Subsequently, an assembler obtains a commitment from a supplier, and this commitment is expressed as an obligation. It is expected that suppliers receive a payment for their components. This requirement specifies the structure of the ObligationToPay obligation (Listing 4), defining that it will be activated by an order message and that it will be deactivated with the delivery of the components and also with the payment. A supplier will only deliver the product if the assembler has the obligation to pay for it (Listing 3).

```
<Transition id="orderTransition" from="as3"to="as4" message-ref="order"/>
<Transition id="deliveryTransition" from="as4" to="as5" message-ref="delivery">
   <ActiveNorms>
      <Norm ref="ObligationToPay"/></ActiveNorms>
</Transition>
```

Listing 3. ObligationToPay usage.

```
<Obligation id="ObligationToPay" abstract="true">
   <Assignee role-ref="assembler" role-instance="$assembler.instance"/>
   <Activations>
     <Element ref="orderTransition" event-type="transition_activation"/>
   </Activations>
   <Deactivations>
    <Element ref="payingTransition" event-type="transition_activation"/>
   </Deactivations>
</Obligation>
```

Listing 4. The ObligationToPay specification.

III.2 THE EXTENDS OPERATOR

As laws can be defined as abstract, with some elements to be further detailed, at implementation time we still need instruments to describe the modifications to turn laws concrete. The *extends* operator changes an abstract element into a concrete one and it cannot leave any element unspecified. This operator is similar to the specialization operation in object-oriented languages (e.g. extends in Java). Basically, the *extends* operator reuses the description of law elements and includes any modifications that are necessary to customize the law element to users' needs, including the redefinition of law elements. For example, this operator can include new activation references, new action elements, and new norm elements and can also superpose any element that was previously specified.

It is also possible to use the *extends* operator to extend a non abstract description. When the *extends* operator is used you should instantiate all abstract slots, that is, you cannot leave any element unspecified. An example of the *extends* operator usage is given in the specification of a down payment in a negotiation protocol. According to the depicted scenario [7], suppliers wishing to protect themselves from defaults will bill agents immediately for a portion down of the cost of each order placed. This down payment is implemented by the action SupplierPayment (Listing 5). Notice that we have added a definition regarding the existence of an action in the context of the obligation definition.

```
<Obligation id="ObligationToPay2005" extends="ObligationToPay">
<Actions>
   <Action id="supplierPayment" class="norm.actions.SupplierPayment">
      <Element ref="orderTransition" event-type="transition_activation"/></Action>
</Actions></Obligation>
```

Listing 5. ObligationToPay extension.

III.3 IMPACTS ON THE M-Law MEDIATOR

Since M-Law works based on XMLaw specification, its lifecycle will be directly impacted by evolution scenarios or even inconsistencies proposed during the interpretation process. The adapted MLaw lifecycle to support static law evolution is composed of five states: idle; interpreting; extending; inconsistent; and running.

The IDLE state is the initial state and it means that the mediator has not started yet. To evolve from the IDLE state to the INTERPRETING state, it is necessary to provide the first set of law elements to be used by the mediator. The INTERPRETING is the state in which the mediator interprets the law definition and maps it to the descriptor model. In the RUNNING state the mediator enforces messages and it is open for services. The INCONSISTENT state is achieved after an attempt to evolve the law with a new element that is inconsistent with the previous specification (e.g. has reference to non-existent elements). The EXTENDING state is a transient state reached after the change request for evolution. After receiving and proceeding with the changes the mediator checks to see if there is any reference for non-existent elements, if so, the state changes to INCONSISTENT and if not, it changes to RUNNING.

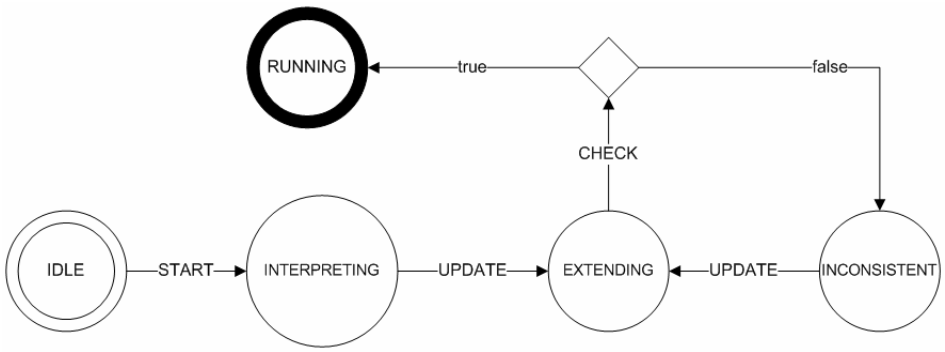

Figure 2. The M-Law mediator lifecycle.

IV. DESIGNING LAW EVOLUTION

In this section, we introduce a design approach that may be used by a law enforcement mechanism to support law extension. The solution presented is not intended to describe a particular concrete design or implementation. Instead, the solution provides how a

general arrangement of classes and objects solves it. We depict the class diagram that contains the interfaces that must be implemented by solutions that support this approach.

During configuration time, an XMLaw specification is interpreted to verify if it is well-formed. The interpreter also maps the law specification to an execution model that will be used at runtime. With the proposal of refinement operators, this process occurs in two steps. First, the basic law is verified and the structure of the execution model is created. Second, the extensions are interpreted and any extension point that was left unspecified is refined. These two steps permit the verification of some construction rules. For example, if a law is defined as concrete, it cannot leave any element to be further refined. All elements must be fully implemented; otherwise, the interpreter will indicate an error.

The class diagram (Figure 3) is composed of five classes. The *EvolutionManager* class acts as a Facade for evolution operations. It contains the methods add(), change(), update(), and remove(). It can be used for runtime law evolution [5] and design time law evolution [4]. We have applied this design in both scenarios. For our context, we will explain the method update(), which deals with evolution in design time. The *Descriptor* class represents the object model of the elements of the XMLaw conceptual model. For example, the scene element in XMLaw is represented by the *SceneDescriptor* class. Its main responsibility is to keep all the information regarding the element and to create execution instances of its descriptor through the createExecution() method.

An object that implements the *Execution* interface is an instance of an element represented by a *Descriptor* object. For example, a scene may be instantiated many times and even various scenes may be running at the same time (various auctions running in parallel, for instance). Each instance (*Execution*) has to keep its instance attributes and control its lifecycle. The *Execution* interface defines all the callback operations needed by the *ExecutionManager* to control instances. The *ExecutionManager* manages all the execution

instances controlling the life cycle. Finally, *DescriptorManager* is in charge of keeping control over all the descriptors being used by the law specification as well as all the cross-references among those descriptors.

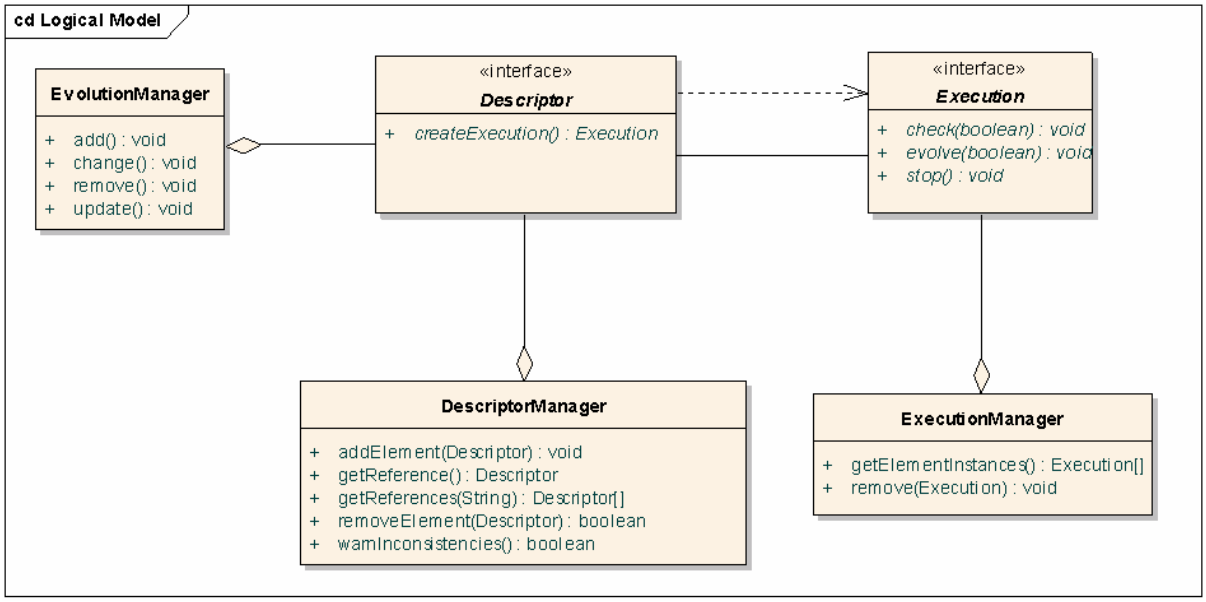

Figure 3. Law evolution design support.

In the context of the implementation of refinement operators, we only need to change the law descriptors because there is no execution element instantiated. In this sense, we need only to understand how the *EvolutionManager* updates one law descriptor. Those five classes interacts to provide the main update() operation. First of all, when we extend one law element we expect that it exists before the change. If there is no element in the *DescriptorManager* context, an exception is thrown. If it exists, its definition instance is returned to be update. During this update, the *EvolutionManager* verifies if all the elements referenced by the new element really exists, if so, the *EvolutionManager* updates the instance of the Descriptor, which represents the new element. Finally, if it was abstract, it is turned to concrete during this process. From this moment, it is possible to create instances (*Execution* instances) of the new updated element.

V. CASE STUDY: THE SELIC APPLICATION (NEGOTIATION SCENARIO)

The Central Bank of Brazil (CCB) regulates and supervises the national financial system. This experiment is running based on the SELIC system requirements [6]. SELIC is the central depository of securities issued by the CCB. It also settles transactions with these securities. Investment banks, savings banks, dealers and many other institutions that integrate the financial system participate in SELIC as holders of custody accounts. In December 2004, the system was composed of 4,900 participants (or agents). SELIC system is clearly a system that has a central entity (CCB) that mediates and controls the interaction among the other entities. We have, then, specified the laws that the institutions must follow using XMLaw, and we have used M-Law as a mediator that control the interactions. We have implemented a prototype of a subset of the actual SELIC system for doing this experiment. The experiment was performed with 10 agents representing different financial institutions and 1 mediator agent (the MLaw).

The purpose of this case study was to understand how we could design interaction laws that could be reused in different contexts. We chose the domain of negotiation and we proposed an abstract law called NegotiationScene (Listing 6) that will be customized for two different auction contexts called OFPUB and LEINF.. The general flow of messages (Figure 4) includes an announcement message to signal that the auction is open to the financial institutions (FI) to send their proposals. Proposals with errors are rejected by the system. The securities negotiation ends in a predetermined period after the beginning of the negotiation. At the end, the system publishes the negotiation results. Listing 6 depicts the code for this scene.

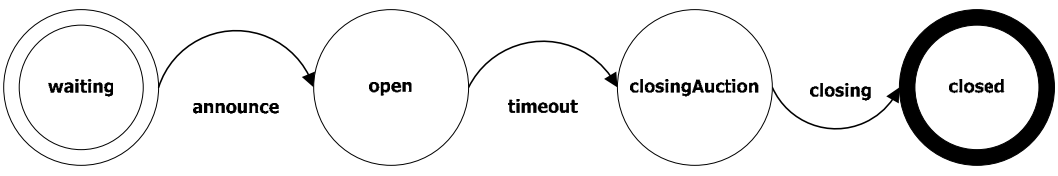

Figure 4. Negotiation Scene Interaction Protocol

```
<Scene id="NegotiationScene" abstract="true">
   <Creators>
      <Creator role="SELIC"/></Creators>
   <Entrance>
      <Participant role="SELIC" limit="1"/>
      <Participant role="FI" limit="N"/></Entrance>
<Protocol id="negotiation" abstract="true">
   <States>
     <State id="waiting" type="initial"/>
     <State id="open" type="execution"/>
     <State id="closingAuction" type="execution"/>
     <State id="closed" type="success"/></States>
   <Transitions>
      <Transition id="announce" from="waiting" to="open" ref="CFP"
                  event-type="action_activation"/>
      <Transition id="timeout" from="open" to="closingAuction" ref="negotiationTimeout"
                  event-type="clock_tick "/>
      <Transition id="closing" from="closingAuction" to="closed" ref="FINISH"
                  event-type="action_activation"/>
   </Transitions></Protocol>
<Actions>
   <Action id="CFP" abstract="true">
      <Element ref="NegotiationScene" event-type="scene_creation"/></Action>
   <Action id="FINISH" abstract="true">
       <Element ref="timeout" event-type="transition_activation"/></Action>
   <Action id="ERR" class="message.ERR" abstract="true"/>
</Actions>
<Clocks>
   <Clock id="negotiationTimeout " type="regular" abstract="true">
   <Activations>
      <Element ref="NegotiationScene"
            event-type="scene_activation"/></Activations></Clock>
</Clocks>
</Scene>
```

Listing 6. Abstract Negotiation Scene implementation

The negotiation protocol is defined as abstract. The expected messages were not specified in the negotiation process: this will be done at instantiation time. Every message sent by a FI must be checked by a constraint that implements some validation rules according to the specific auction requirements. An error handling policy is implemented to avoid that incorrect messages are propagated throughout the negotiation. This policy is implemented by the action ERR that sends an error notification to the sender. Some actions are defined as abstract to be implemented in the instance. There is also a clock element that controls the time limit of the negotiation. The actual period of negotiation will be determined at instantiation time.

The first instance is called OFPUB (Listing 7, Figure 5). This auction is used to promote the sell of bonds by National Treasury to FIs. This auction is an instance of negotiationScene. Every day, the system configures (**CONFIG** action) and communicates to

participants (**CFP**) that the auction scenario is open to the financial institutions to send their proposals (**PROPOSAL**). Proposals with errors are rejected by the system. The message **PROPOSAL** was defined and its associated restriction (**checkProposal**) was implemented as Java component to fulfill the bussiness rules of the **OFPUB** auction scenario. This restriction would fire the action **ERRORFPUB** that will notify the message's sender of any problem with its message. The actions **CONFIG**, **OFPUB_CFP** and **OFPUB_RESULT** are responsible for the active behavior of SELIC and were implemented as Java components. The clock **timeoutOFPUB** indicates the end of the negotiation and was configured to fire a clock tick after four hours (**864000ms**).

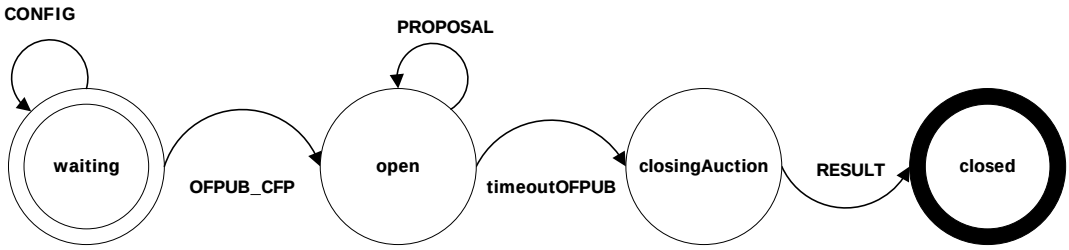

Figure 5. OFPUB Protocol extens Negotiation Protocol

```
<Scene id="OFPUB" extends="NegotiationScene">
   <Protocol id="OFPUBnegotiation" extends="negotiation">
     <Messages>
      <Message id="PROPOSAL"/></Messages>
     <Transitions>
      <Transition id="configuration" from="waiting" to="waiting" ref="CONFIG"
                  event-type="action_activation"/>
      <Transition id="receivingProposal" from="open" to="open" ref="PROPOSAL"
                  event-type="message_arrival">
       <Constraints>
       <Constraint id="checkProposal" class="constraint.checkProposal"/></Constraints>
     </Transition></Transitions>
   </Protocol>
   <Actions>
   <Action id="CONFIG" class="ofpub.CONFIG">
      <Element ref="OFPUB" event-type="scene_creation"/></Action>
   <Action id="OFPUB_CFP" extends="CFP" class="ofpub.CFP"/>
   <Action id="RESULT" extends="FINISH" class="ofpub.RESULT"/>
   <Action id="ERROFPUB" extends="ERR">
     <Element ref="checkProposal" event-type="constraint_activation"/></Action>
   </Actions>
   <Clocks>
      <Clock id="timeoutOFPUB" tick-period="864000" extends="negotiationTimeout"/>
   </Clocks>
</Scene>
```

Listing 7. OFPUB Scene implementation extends NegotiationScene

The second instance is called LEINF (Listing 8, Figure 6).. This auction is used to promote the exchange of securities in an open market. This auction is an instance of **negotiationScene**. Every day, the system communicates (**LEINF_CFP**) that the auction scenario is open for the FIs to send their **BUY** and **SELL** proposals. Proposals with errors are rejected by the system. The messages **BUY** and **SELL** were defined and their associated restrictions (**checkBUY** and **checkSELL**) were implemented as Java components to fulfill the business rules of the **LEINF** auction scenario. Those restrictions would trigger the action **ERRORLEINF** that will notify the message's sender of any problem with its message. The actions **LEINF_CFP** and **LEINF_RESULT** are responsible for the active behavior of SELIC and were implemented as Java components. The clock **timeoutLEINF** indicates the end of the negotiation and was configured to fire a clock tick after six hours (**1296000ms**),

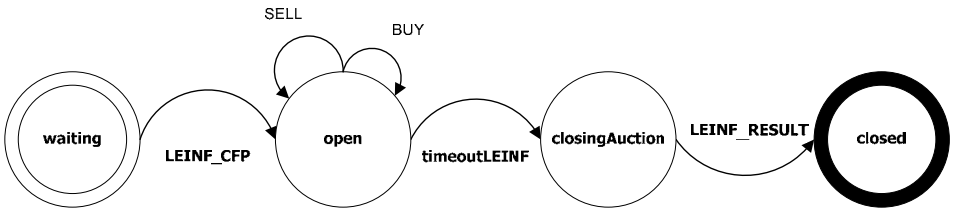

Figure 6. LEINF Protocol extends NegotiationScene

```
<Scene id="LEINF" extends="NegotiationScene">
<Protocol id="negotiationLEINF" extends="negotiation">
   <Messages>
      <Message id="BUY"/>
      <Message id="SELL"/></Messages>
   <Transitions>
      <Transition id="receivingBUY" from="open" to="open" ref="BUY"
                  event-type="message_arrival">
          <Constraints>
             <Constraint id="checkBUY" class="constraint.checkBUY"/>
          </Constraints></Transition>
      <Transition id="receivingSELL" from="open" to="open" ref="SELL"
                  event-type="message_arrival">
          <Constraints>
             <Constraint id="checkSELL" class="constraint.checkSELL"/>
          </Constraints></Transition></Transitions>
</Protocol>
<Actions>
   <Action id="LEINF_CFP" extends="CFP" class="leinf.CFP"/>
   <Action id="LEINF_RESULT" extends="FINISH " class="leinf.RESULT"/>
   <Action id="ERRORLEINF" extends="ERR">
       <Element ref="checkBuy" event-type="constraint_activation"/>
       <Element ref="checkSELL" event-type="constraint_activation"/></Actions>
<Clocks>
   <Clock id="timeoutLEINF" tick-period="1296000" extends="timeoutNegotiation">
</Clocks>
</Scene>
```

Listing 8. LEINF Scene implementation extends NegotiationScene

This case study illustrated the application of refinement operators in a real-world negotiation scenario of SELIC, and two scenes (OFPUB and LEINF) were specialized by the same basic scene. In this example, it was possible to design a more general law that was customized into two different contexts. Refinement operators have simplified and reused the design and the implementation of those law elements.

VI. RELATED WORK

Minsky [2, 15] proposes a coordination and control mechanism called law-governed interaction (LGI). It provides a language to specify laws and it is concerned with architectural decisions to achieve a high degree of robustness. In contrast, XMLaw provides an explicit conceptual model and focuses on different concepts such as Norms and also interaction extensibility support. Ao and Minksy [2] propose an approach to enhance LGI with the concept of policy-hierarchy to support that different internal policies are formulated independently of each other, achieving flexibility support by this means. Different from our approach, Ao and Minsky consider confidentiality as a requirement for their solution. The goal of the extensions that we have presented until now is to support open system law maintenance instead of flexibility for confidentiality purposes.

COSY [11] views a protocol as an aggregation of primitive protocols. Each primitive protocol can be represented by a tree where each node corresponds to a particular situation and transitions correspond to possible messages an agent can either receive or send, i.e., the various interaction alternatives. In AgenTalk [13], protocols inherit from one another. They are described as scripts containing the various steps of a possible sequence of interactions. Beliefs also are embedded into scripts. Koning and Huget [12] deal with the modeling of interaction protocols for multi-agent systems, outlining a component-based approach that improves flexibility, abstraction and protocol reuse. All of the approaches listed in this

paragraph are useful instruments to promote reuse; they can be seen as instruments for specifying extendable laws.

Singh [20] proposes a customizable governance service, based on skeletons. His approach formally introduces traditional scheduling ideas into an environment of autonomous agents without requiring unnecessary control over their actions, or detailed knowledge of their designs. Skeletons are equivalent to state-based machines and we could adapt and reuse their formal model focusing on the implementation of extensions. But [20] has its focus on building multi-agent systems instead of law monitoring and enforcement.

VII. CONCLUSIONS

While analyzing the open software system domain, it is possible to design part of the open system evolution. If a desired characteristic of a system is long-term stability, then the challenge for developers is to deliver a product that identifies the general aspects of the open MAS that will not change. We are addressing the problem of constructing governance mechanisms that ensure that agents will conform to a well defined customizable specification. Our main goal is to contribute on the engineering level about how we can productively define and reuse laws. We have applied and adapted some object oriented concepts to improve law maintainability in XMLaw. XMLaw combines extensible interaction specification with Java components to instruct how governance mechanisms should enforce the expected behavior. We are also contributing with the study on how to engineer governance mechanisms development. With the refinement operators, we support the design of law elements for extension. We are aware of possible consistency problems when redefining or extending laws. We are dealing with this problem through the definition of a formal framework that enables us to check possible inconsistencies. However, a deeper discussion is beyond the scope of this paper.

REFERENCES

1. Agha, G. A. (1997) Abstracting Interaction Patterns: A Programming Paradigm for Open Distributed Systems, In (Eds) E. Najm and J.-B. Stefani, Formal Methods for Open Object-based Distributed Systems IFIP Transactions, Chapman & Hall.

2. Ao, X. and Minsky, N. (2003). Flexible Regulation of Distributed Coalitions. In Proc. of the 8th European Symposium on Research in Computer Security (ESORICS).

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

4. Carvalho, G.; Lucena, C.; Paes, R.; Briot, J-P. (2006) Refinement Operators to Facilitate the Reuse of Interaction Laws in Open Multi-Agent Systems, International Workshop on Software Engineering for Large-scale Multi-Agent Systems (SELMAS'06), 5th, at ICSE 2006, Shanghai, China. In: Proceedings of the Fifth International Workshop on Software Engineering for Large-scale Multi-agent Systems, pp. 75-82.

5. Carvalho, G. et al; Dynamic Law Evolution in Governance Mechanisms for Open Multi-Agent Systems. Workshop on Software Engineering for Agent-oriented Systems (SEAS 2006), 2nd, In: Lucena, C. et al (eds.) Proceedings of the II Workshop on Software Engineering for Agent-oriented Systems (SEAS 2006), pp. 83-94, October 17th, 2006.

6. Case Study Requirements – SELIC application scenario - http://www.bcb.gov.br/?SELIC. Last Visit <Jan/12/2007>

7. Collins, J; Arunachala,R; Sadeh,N; Eriksson,J; Finne,N; Janson,S. (2005) The Supply Chain Management Game for the 2005 Trading Agent Competition. CMU-ISRI-04-139. http://www.sics.se/tac/tac05scmspec_v157.pdf

8. Esteva, M. (2003) Electronic institutions: from specification to development, Ph.D. thesis, Institut d'Investigació en Intelligència Artificial, Catalonia - Spain.

9. Fredriksson M. et al. (2003) First international workshop on theory and practice of open computational systems. In Proceedings of twelfth international workshop on Enabling technologies: Infrastructure for collaborative enterprises (WETICE), Workshop on Theory and practice of open computational systems (TAPOCS), pp. 355 - 358, IEEE Press.

10. Gamma, E.; Johnson, R.; Helm, R.; Vlissides, J. (1995) Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley.

11. Haddadi, A. (1996) Communication and Cooperation in Agent Systems: A Pragmatic Theory, volume 1056 of Lecture Notes in Computer Science. Springer Verlag.

12. Koning, J.L. and Huget, M.P. (2000) A component-based approach for modeling interaction protocols. In H. Kangassalo and E. Kawaguchi, editors, 10th European-Japanese Conference on Information Modeling and Knowledge Bases, Frontiers in Artificial Intelligence and Applications.IOS Press
13. Kuwabara, K; Ishida, T; and Osato, N. (1995) AgenTalk: Coordination protocol description for multiagent systems. In First International Conference on MultiAgent Systems (ICMAS-95), AAAI Press.
14. LGI homepage. http://www.moses.rutgers.edu/ . Visited in 12/01/2005.
15. Minsky, N. H.; and Ungureanu V. (2000) Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems, ACMTrans. Softw. Eng. Methodol. 9 (3) 273–305.
16. Paes, R. B.; Carvalho G. R.; Lucena, C.J.P.; Alencar, P. S. C.; Almeida H.O.; Silva, V. T (2005). Specifying Laws in Open Multi-Agent Systems. In: Agents, Norms and Institutions for Regulated Multi-agent Systems (ANIREM), AAMAS2005.
17. Paes, R.B; Lucena, C.J.P; Alencar, P.S.C. (2005). A Mechanism for Governing Agent Interaction in Open Multi-Agent Systems. http://www.les.inf.puc-rio.br/governance/pubs.html
18. Pree,W. (1997) Essential Framework Design Patterns.Object Magazine.
19. Sadeh, N; Arunachalam, R; Eriksson, J; Finne, N; Janson, S. (2003) TAC-03: a supply-chain trading competition, AI Mag. 24 (1) 92–94.
20. Singh, M. P. (1998) A Customizable Coordination Service for Autonomous Agents," Intelligent Agents IV: Agent Theories, Architectures, and Languages, Munindar P. Singh et al. ed., Springer, Berlin, pp. 93-106.
21. Wooldridge, M; Weiss, G; Ciancarini, P. (Eds.) (2002) 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.
22. Zambonelli, F, Jennings, N; Wooldridge, M. (2003) Developing multiagent systems: The gaia methodology, ACM Trans. Softw. Eng. Methodol. 12 (3) 317–370.