Dynamic Law Evolution in Governance Mechanisms for Open Multi-Agent Systems

Gustavo Carvalho¹, Hyggo Almeida², Maira Gatti¹, Glauber Ferreira², Rodrigo Paes¹, Angelo Perkusich², Carlos Lucena¹

¹Laboratório de Engenharia de Software
Departamento de Informática - Pontifícia Universidade Católica do Rio de Janeiro
Rio de Janeiro – RJ – Brazil

²Laboratório de Sistemas Embarcados e Computação Pervasiva
Centro de Engenharia Elétrica e Informática
Universidade Federal de Campina Grande – Campina Grande – PB – Brazil
{guga,mgatti,rbp,lucena}@inf.puc-rio.br,
{hyggo,glauber,perkusic}@dee.ufcg.edu.br

Abstract. This paper introduces the support for dynamic law evolution in the XMLaw governance approach. This approach provides mechanisms for managing unpredicted law changes on the fly. We propose a generic software design for developing mechanisms to support dynamic law evolution. The implementation of the proposed design is also presented considering two possible underlying software solutions: OSGi and COMPOR CMS. Finally, a negotiation scenario is used to exemplify how this approach can be useful for governance mechanism maintenance.

1. Introduction

Open Multi-Agent Systems (open MAS) run on dynamic environments composed of autonomous distributed agents that may enter and leave the environment at will. Such agents are developed by different engineering teams, and there is no centralized control over the development of them [Agha, 1997]. This may cause conflicting interests among agents [Fredriksson et al, 2003], requiring mechanisms that guarantee the interaction rules of the Multi-Agent System as a whole.

Moreover, since agents are often autonomous, they can behave unpredictably, and unforeseen situations can arise. These characteristics may cause scenarios where we must monitor and control the execution of the system to achieve dependability. In this sense, we believe that the specification of open MAS should include laws that define what and when something should happen in it. Such laws are restrictions imposed by the environment to tame uncertainty and to promote open MAS dependability [Minsky et Ungureanu, 2000][Paes et al., 2005]. An open MAS should also include a governance mechanism; that is, a mediator that enforces the law specification [Esteva, 2003] [Minsky et Ungureanu, 2000] [Paes et al.,2005b].

Interaction protocols that govern interaction among agents, and other law elements, like constraints, permissions and prohibitions, are specified considering the requirements of the open MAS. To this end, XMLaw [Paes et al., 2005] defines a language to describe the laws that govern an open MAS. During the execution of such systems, the middleware M-Law [Paes et al.,2005b] can be used as the mediator to
support the monitoring of agents and verification of compliance with the laws defined using XMLaw.

However, the dynamicity inherent to open MAS implies that the interaction laws may frequently be changed. These changes may be required during execution time, and open MAS execution might not be interrupted in some scenarios. Such an evolution typically deals with the addition, removal or change of law elements. Examples of changes are the exclusion of a permission that enables the entrance of an agent in a negotiation, the inclusion of a payment obligation that an agent has after it buys a product in a negotiation, and finally the changes in the negotiation protocol to include two new steps.

This paper proposes an approach to support dynamic evolution of laws. We discuss the impact of this proposal on XMLaw conceptual elements. This includes an analysis of how to change their execution lifecycles and how to improve the M-Law mediator in order to support changes in the current specification at runtime. The improvements in M-Law have been implemented to enable a continuous compliance verification of software agents inhabiting open MAS.

We discuss how to apply two potential software techniques to support dynamic law evolution in governance mechanisms: OSGI (Open Services Gateway Initiative) and COMPOR CMS (Component Model Specification). OSGI is a specification for service oriented development for networked devices in Java [OSGI Alliance, 2006]. COMPOR CMS is a component based model for developing software supporting dynamic unanticipated evolution [Almeida et al, 2006]. In our context, such techniques will allow changing any part of the law specification, removing and adding law elements, even at runtime.

The remainder of this paper is organized as follows. Section 2 describes background concepts including details on XMLaw and M-Law. Section 3 details concerns regarding the dynamic evolution of laws, including our general strategy and the analysis of the structural and behavioral impact of this proposal on the lifecycle of law elements. To illustrate this analysis we use a negotiation scenario. Section 4 introduces the details required by a law enforcement mechanism to support dynamic law evolution. In Section 5 we describe how COMPOR CMS and OSGI can be integrated to M-Law to support dynamic evolution of laws. Related work is discussed in Section 6. Finally, concluding remarks are presented in Section 7.

**2. Governing Interactions in Open Multi-Agent Systems**

Law-governed architectures are designed to guarantee that the specifications of open systems will be obeyed. M-Law is an example of law-governed infrastructure provided to agent developers [Paes et al, 2005b]. M-Law works by intercepting messages exchanged between agents, verifying the compliance of the messages with the laws and subsequently redirecting the message to the real addressee, if the laws allow it. If the message is not compliant, the mediator blocks the message and applies the consequences specified in the law. This infrastructure, whenever necessary, can be extended to fulfill open system requirements or interoperability concerns. M-Law architecture is based on a pool of mediators that intercept messages and interpret the previously described laws (Figure 1). Mediators monitor the exchange of messages
between agents. As more clients are added to the system, additional mediator instances can be added to improve throughput.

![Diagram of M-Law architecture](image)

**Figure 1 - M-Law architecture**

M-Law has been built to support law specification using XMLaw [Paes et al, 2005a]. XMLaw is the description language used to configure the M-Law mediator by representing the interaction rules of an open system. These rules are interpreted by M-Law that analyzes the compliance of software agents with interaction laws at runtime [Paes et al, 2005b]. XMLaw represents the structure and the relationships between law elements.

A law specification is a description of law elements. Law elements are interrelated in a way that it is possible to specify interaction protocols using time restrictions, norms, or even time sensitive norms.

The conceptual model that defines the law specification 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. Every scene specifies one protocol. Because protocols define the interaction among the agents, different protocols should be specified in different scenes. Scenes also specify which agent role has permission to create scene instances.

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 and also temporal elements, to provide a dynamic configuration for the allowed behavior of agents in an open system.

Norms prescribe how the active distributed software agents ought to behave, and specify how they are permitted to behave and what their rights and duties are. The mediator keeps information about the set of activated norms to verify the compliance of software agents, the set of deactivated norms and any other data regarding system execution. There are three types of norms in XMLaw: obligations, permissions and prohibitions. A Norm can be activated and deactivated by events. For example, 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). Norms also define the agent role that owns it through the attribute Assignee. Norm events and status (activated or deactivated) are referenced by other elements. For instance, as a consequence of the relationship between norms and transitions, it is possible to specify which norms must be activated or deactivated for firing a transition, i.e., a transition could only fire if the sender agent has a specific norm. Norms also have constraints and actions associated with them.
Constraints are restrictions over norms or transitions and generally specify 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. Constraints can be used for describing the allowed values for specific attributes. Constraints are defined inside the Transition or Norm 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 received values are valid. For instance, a constraint can verify if the date expressed in the message is valid; if it is not, the message will be blocked.

Laws may be time sensitive, e.g., although an element that is active at time $t_1$, it might not be active at time $t_2$ ($t_1 < t_2$). XMLaw provides the Clock element to take care of the timing aspect. Clocks represent time restrictions or controls and they can be used to activate other law elements. Clocks indicate that a certain period has elapsed producing clock-tick events. Once activated, a clock can generate clock-tick events. Clocks are activated and deactivated by law elements. Both are referenced to other law elements.

3. Analyzing the Impact of Law Evolution at Runtime

One of the main issues in developing, deploying, and monitoring open Multi-Agent Systems is to deal with dynamicity and unpredictability. During the execution of open MAS changes not predicted at design time could be required. Such unanticipated changes must be managed at runtime, since the execution of the system cannot be interrupted in some scenarios including banking systems, telecommunication systems, military systems, e-commerce systems, among others.

Law evolution typically deals with the addition, removal or changing of law elements. Some possible evolution scenarios are: the definition of new agent roles and the redefinition of existing ones; the addition, removal, or changing of law elements (e.g. norms, clocks, and constraints); the specification of new protocols and changes in existing ones; and the redefinition of scenes.

Below, we discuss the impacts of this scenario over the mediator lifecycle and over other law elements.

3.1. The Impact over M-Law Mediator Lifecycle

Since M-Law works based on XMLaw specification, its lifecycle will be directly impacted by potential unpredicted dynamic evolution scenarios. The adapted M-Law lifecycle to support dynamic law evolution is composed of five states (Figure 2): IDLE, RUNNING, EVOLVING, INCONSISTENT, and FINAL.

The IDLE state is the initial state and it means that the mediator has not started yet. The RUNNING is the state in which the mediator receives messages and enforces the laws that have been previously defined. To evolve from the IDLE state to the RUNNING state, it is necessary to provide the first set of law elements to be used by the mediator. In the FINAL state the mediator does not enforce any messages and it is closed for services. A mediator can receive a new request for law evolution in the RUNNING state and in the INCONSISTENT state. 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 EVOLVING state is a transient state reached after any request for evolution (add, change, or remove). The mediator saves all the intercepted messages in a queue for future processing. 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 - Mediator lifecycle](image)

### 3.2. The Impact Analysis Pattern over Law Elements

This section presents the impact analysis pattern for the evolution of laws at runtime through a simple scenario that allows exploiting the main concerns regarding our proposal. The scenario is about a loan process in which a client asks (request) for loan application, receives the offer (inform) from a loan provider, and replies the offer with a refuse or agree message. Some clients have the permission to loan any amount of money with no constraints (requestSpecial). These steps are shown in Figure 2. We took four evolution examples covering the three types of evolution: insertion, removal and changing. Each example is followed by the analysis of the structural and behavioral impacts.

In order to evolve this scenario, all the elements that make explicit reference to another element also might be changed. Thus, the change may be structurally analyzed by two points of view: definition and references. In other words, it may be necessary to change the definition of the element being added, removed or changed, and also we need to search for the references that have been made to this element by other law elements.

![Figure 2 - Original definition of the loan process](image)
Considering the behavioral impacts in the system execution, the main questions that arise are: which are the elements already in execution that would have to be updated because of this evolution? And which ones would not? Is there any condition, or all permissions already in execution, for e.g., should be removed from the agents that acquired it? The behavioral impacts on other law elements not considered in the evolution scenarios that are executing are out of scope of this article.

### 3.2.1 Evolution Scenario 1: Permission removal and Constraint insertion

The loan providers have observed that many clients loaned and they did not pay the money back. One reason was that their earnings were not enough to pay for the loan. Then, the loan providers have decided to evolve the laws removing the permission and inserting a constraint that imposes a limit for the loan based on client’s earnings. If the limit is satisfied, the constraint will let the transition to be activated. Figure 3 shows this new scenario.

![Figure 3 - Permission removal and Constraint insertion](image)

In order to evolve the law specification to achieve those changes, it would be necessary to analyze the structural impacts. Table 1 shows the structural impact over the elements by the permission removal and constraint insertion. A permission element is specified in the context of the scene. If a permission definition is removed from the specification, the Scene element would have its definition changed. It would also be necessary to change the transition’s reference to the permission.

Considering the Constraint insertion, the structural impacts would be analogous to the Permission removal since it is defined in the context of the Scene and is also referenced in the Transition, i.e., the Constraint will have to be executed when the message that activates the transition is sent.

<table>
<thead>
<tr>
<th>Element</th>
<th>Structural impact : Definition</th>
<th>Structural impact : Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Permission A</td>
<td>Scene</td>
<td>Transition</td>
</tr>
<tr>
<td>Constraint(limit)</td>
<td>Scene</td>
<td>Transition</td>
</tr>
</tbody>
</table>

Analyzing the behavioral impacts in the system execution in this scenario, we may consider that all the permissions in execution will be removed from the agents in all scenes executions of this scene definition and the constraint inserted will have to be enforced.

### 3.2.2 Evolution Scenario 2: Permission and Clock insertion

Another possible scenario is related to permission and clock insertion. In this case, the loan provider would allow the client to pay its debts with a less interest until a period of time (in this case, thirty minutes). If this time expires, the client is not allowed to take the loan anymore. Figure 4 shows this new scenario.
Figure 4 - Permission and Clock insertion

In this scenario, there are two new elements that have to be inserted in the XMLaw specification: Permission and Clock. The Permission B will impact directly on the definition of the Scene. The other impact will be on the reference of the Transition that will activate this permission and on the reference of the Transition that will only be activated if the agent has permission gave a priori. Considering the structural impacts of the Clock insertion, it is analogous to the permission insertion. Moreover, as the Clock will be activated by the Permission B, there is no impact on the previous definition before starting this evolution.

Table 2 – Structural impacts: Permission and Clock insertion

<table>
<thead>
<tr>
<th>Element</th>
<th>Structural impact: Definition</th>
<th>Structural impact: Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Permission B</td>
<td>Scene</td>
<td>Transition</td>
</tr>
<tr>
<td>Clock (30min)</td>
<td>Scene</td>
<td>Permission B</td>
</tr>
</tbody>
</table>

Analyzing the behavioral impacts in the system execution, the one making the evolution have to concern about the behavior of the elements already in execution. Following the pattern for evaluating the behavioral impacts, this evaluation will imply in the change of the element that were previously instantiated and the procedure is to verify a condition if it is feasible to stop the execution of that element. It will be done by the method `stopExecution(Condition)` that means the instances in execution satisfying certain conditions and that refers or are affected by the evolution should be stopped. In this scenario, we consider that new elements will be created and instantiated according to the new description and the oldest will be updated with the new elements.

3.2.3 Evolution Scenario 3: Clock change

This scenario includes the change of one Clock that has interval of 30 minutes to one that has interval of 60 minutes. The clock’s identification will not change. As we have defined, this change will imply in one removal and one insertion of the clock element. Considering the structural impacts of the Clock removal and insertion, the scene context will be updated. The references will be kept and the Clock will continue to be referenced by the Permission B, that already exists and so there is no impact on reference update.

Figure 5 - Change on the clock definition

Table 3 – Structural impact: Clock change

<table>
<thead>
<tr>
<th>Element</th>
<th>Structural impact: Definition</th>
<th>Structural impact: Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock</td>
<td>Scene</td>
<td>-</td>
</tr>
</tbody>
</table>
3.2.4 Evolution Scenario 4: State and Transition insertion

The fourth evolution example occurred because of a change in the national laws. The country where the loan providers are placed has created a law stating that every loan must occur through a signed contract between the loan provider and the client. Then the law of the system also has to evolve to support the new regulation.

Then, the final state has changed to a regular state and two other states were added to regulate the signed contract. The first added transition is a communication from the loan provider to the client informing the terms of the contract. Then, the client may confirm or refuse the terms. The structural impacts are summarized in Table 4.

Table 4 – Structural impact : Protocol evolution

<table>
<thead>
<tr>
<th>Element</th>
<th>Structural impact : Definition</th>
<th>Structural impact : Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>State S4</td>
<td>Protocol</td>
<td>Transition</td>
</tr>
<tr>
<td>State S5</td>
<td>Protocol</td>
<td>Transition</td>
</tr>
<tr>
<td>State S6</td>
<td>Protocol</td>
<td>Transition</td>
</tr>
<tr>
<td>State Unsuccess end</td>
<td>Protocol</td>
<td>Transition</td>
</tr>
<tr>
<td>Transition agree</td>
<td>-</td>
<td>State</td>
</tr>
<tr>
<td>Transition hire</td>
<td>Protocol</td>
<td>State</td>
</tr>
<tr>
<td>Transition confirm</td>
<td>Protocol</td>
<td>State</td>
</tr>
<tr>
<td>Transition refuse</td>
<td>Protocol</td>
<td>State</td>
</tr>
</tbody>
</table>

Figure 6 - State and Transition insertion

Once again, the one making the evolution have to concern about the behavior of the elements already in execution. Following the pattern for evaluating the behavioral impacts. This evaluation will imply in the change of the element that has been previously instantiated and the procedure is to verify a condition if it is feasible to stop the execution of that element. It will be done by the method stopExecution( Condition ) that means that the instances in execution satisfying certain conditions and that refers or are affected by the evolution should be stopped.

4. Design for Dynamic Law Evolution

In this section, we introduce a design approach that may be used by a law enforcement mechanism to support dynamic law evolution, considering the evolution analysis pattern of the XMLaw governance mechanism (proposed in Section 3). 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.

![Class Diagram](image)

**Figure 7 - Law evolution pattern**

The class diagram is composed of five classes. **EvolutionManager** acts as a Facade for dynamic evolution operations. It contains the methods **add()**, **change()**, and **remove()**. 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 (not shown in the Figure). Its main responsibility is to create execution instances of the 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 lifecycle. 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.

Those five classes interacts to provide the three main operations concerning evolution: **add()**, **change()**, and **remove()**. In the **add()** operation, the designer wishes to add a new element specification in the law at runtime. Then, first the **EvolutionManager** verifies if all the elements referenced by the new element really exists, if so, the **EvolutionManager** creates an instance of a **Descriptor**, which represents the new element. Finally, the **EvolutionManager** updates the **DescriptorManager** with the just created **Descriptor**. From this moment, it is possible to create instances (**Execution** instances) of the new added element.

The **remove()** operation occurs when the law designer removes an element from the law specification. In this case, the **EvolutionManager** calls the **removeElement()** method in the **DescriptorManager**. If the descriptor was successfully removed, the
method returns true and then the EvolutionManager gets all the elements that references the removed element. For each referenced element, the EvolutionManager interacts to the ExecutionManager to get the references to the instances of the referenced elements (Execution). Then, for each execution, the EvolutionManager calls the evolve() method. Each execution checks the parameter condition, which verifies if the current execution is able to be evolved. In its last task, the EvolutionManager verifies if there is any lost reference in the descriptor manager caused by ExecutionManagers that didn’t evolve because of a specific condition and notifies the designer through warnings.

The last operation for evolution is the change(). This operation is simply composed of the removal and inclusion of law elements.

5. Underlying Software Infrastructures

We have considered two software infrastructures to develop the support for dynamic evolution of laws previously described: COMPOR CMS and OSGI.

COMPOR Component Model Specification (CMS) is a component based model to develop software supporting dynamic unanticipated evolution. According to CMS, a component based system is described as a composition of two kinds of entities: functional components and containers. Functional components are software entities implementing application specific functionalities, making them available by means of services and events. A functional component is not composed of other components, that is, it has no child components. Containers are software entities that implement no application-specific functionalities. A container controls the access to the services and events provided by its child components, which may be functional components or other containers [Almeida et al., 2006].

In order to implement our proposal using CMS, the Descriptors instances described in Section 4 are implemented as functional components or containers, depends on their composition characteristics. For example, since State and Transition are not composed of other elements, they are implemented as functional components; since Protocol is composed of States and Transitions, it is implemented as a container. Following this conventions, law definitions could evolve, since CMS supports the addition, removal, and changes of their entities, even at runtime.

The OSGi specifications define a specification for service oriented programming using Java. It provides a component oriented, computing environment for networked services. OSGi promotes the capability to manage the life cycle of the software components and services, named bundles, in a given device from anywhere in the network. Software bundles can be installed, updated, or removed on the fly [OSGI Alliance, 2006].

In order to apply OSGi to implement our design proposal, each Descriptor of the XMLaw conceptual model is implemented as an OSGi bundle. Bundles for elements that are not composed of other elements (e.g. State and Transition) register their references as services in order to be retrieved. Bundles for elements composed of other elements (e.g. Protocol) retrieves their child elements through the previously registered services. As well as in CMS, this solution allows the evolution of law definitions since Descriptor elements (implemented as bundles) can be installed, updated, or removed on the fly.
Differently from CMS, OSGi does not define a hierarchical structure. This could cause a maintenance problem considering large amounts of deployed bundles at the same level, since Descriptor elements are conceptually represented by a hierarchy. On the other hand, the versioning mechanism provided by OSGi is very useful for managing several versions of a bundle running at the same time. At implementation level, the necessary programming effort is equivalent for both techniques.

6. Related Work

In the context of governance approaches, two important research projects have some similarities with XMLaw and M-Law. Esteva [9] proposes the use of scenes and protocol elements for specifying the agent interaction laws. The time aspect is represented in the Esteva approach as timeouts. XMLaw also includes the concept of actions, which allows execution of Java code in response to some interaction situation. Minsky [5] proposes a coordination and control mechanism called law governed interaction (LGI). This mechanism is based on two basic principles: the local nature of the LGI laws and the 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. Differently from XMLaw, the control is decentralized, with distributed mediators. Further comparison between XMLaw and these works can be found in [Paes, 2005a].

As the original version of XMLaw, these works do not support dynamic evolution of laws. We have found only one work related to governance mechanisms supporting dynamic unanticipated evolution. In this work [Mathieu et al, 2002], the authors propose an approach for dynamic multi-agent organizations that supports the evolution of a multi-agent system by redefining the interaction among agents. The concepts of law, norms, obligations, and clocks are not supported. Agents can be inserted and removed at runtime, and the acquaintance model can be reconfigured, but there is no law enforcement mechanism. In fact, although it provides some kind of dynamic evolution, it does not represent a governance strategy.

7. Conclusions and Future Work

This paper presented an approach for dynamic evolution of laws in open Multi-Agent Systems. We discussed the main issues regarding managing law evolution dynamically, using evolution scenarios for detailing the impact of changing, inserting, and removing law elements on the system specification and execution.

We proposed a generic software design to be used for developing mechanisms to support dynamic law evolution. Furthermore, we discussed how two techniques for dynamic software evolution could be applied for promoting dynamic law evolution: OSGI and COMPOR CMS.

As future work, we plan to define an analytical model for evaluating a priori the viability of dynamic evolution. It is necessary to evaluate which strategy must be taken according to a given evolution scenario: stopping the system or updating laws during runtime? Depending on the number of changes to be performed and how large the system is, evolve it during runtime could be impracticable.
References


