Refinement Operators to Facilitate the Reuse of Interaction Laws in Open Multi-Agent Systems

Gustavo Carvalho, Carlos Lucena, Rodrigo Paes

PUC-Rio - Marquês de São Vicente 225
4º Andar RDC – Rio de Janeiro Brasil
+55-21-25406915 – 137
{guga,lucena,rbp}@inf.puc-rio.br

Jean-Pierre Briot
LIP6 - Univ. Paris 6 - CNRS
8 rue du Capitaine Scott
75015 PARIS, France
Jean-Pierre.Briot@lip6.fr

ABSTRACT
As new software demands and requirements appear, the system and its interaction laws must evolve to support these changes. Languages and models should provide the tools for dealing with this evolution. Poor support for evolution has a negative impact on system maintainability. In this paper, we propose some refinement operators to extend the interaction laws in open multi-agent systems. As an example of this idea, we implemented a customizable application in the supply chain management domain as an open system environment.

Categories and Subject Descriptors
I.2.11 [Distributed Artificial Intelligence]: Multiagent Systems—Multiagent Organizations

General Terms
Design

Keywords
Reuse, law-enforcement, software agents, interaction protocol.

1. INTRODUCTION
Nowadays, one characteristic that is crucial for software in many situations is openness. Open systems are composed of autonomous distributed components that may enter and leave the environment at will, and they may even have conflicting interests [8] because open software systems have no centralized control over the development of their parts [1]. Multi-agent auction systems and virtual enterprises are examples of such open applications [21].

Since open system components are often autonomous, sometimes they behave unpredictably and unforeseen situations arise. We believe that the specification of open multi-agent systems (open MAS) should include laws that define what and when something can happen in an open MAS. Laws are restrictions imposed by the environment to tame uncertainty and to promote open MAS dependability [14][15]. A governance mechanism is the mediator that enforces the law specification. Examples of governance mechanisms are LGI[14], Islander[7] and XMLaw[16]. In this paper, we will use the XMLaw language [15] to map customizable specification of interaction rules into a governance mechanism.

The greater the dependence of our society on open distributed applications, the greater will be the demand for dependable applications and also for new solutions that are variations of previously existing ones. Reuse is one of the most important characteristics while developing systems. When you achieve the reuse of software artifacts, you improve software maintainability and reduce development efforts. To fulfill this goal, one of the challenges of software development is to produce software that is designed to evolve, and so be extended.

Nowadays, we do not have much support for the reuse of law specifications. While developing software in a specific domain, usually there are some similar requirements that could imply in similar solutions. If you do not clearly modularize and reuse the solutions you probably will increase development efforts.

We will give a simple example that regulates the interaction between two communities. In order to specify this example, two groups of law elements were generated by copying/pasting the specification of one community to the other. We borrowed this approach from the LGI homepage example [13]. We can specify it in XMLaw (Figure 1) without considering any support to extend or configure a basic definition that can be reused for both communities. In this case, we can observe that copying / modifying / pasting specifications derives a semi-identical specification with few peculiarities. In Figure 1, the code is practically the same between the two definitions; the only difference is the id and the name of some law elements (organization and scene) and also the constraint that is used twice in each scene. Using refinements, we could easily have a basic description of this law and with little customization effort both specifications could be proposed.

As the example showed, open MAS should be specified and developed to facilitate extensions and law-governed approaches should also present a solution to this concern. As open MAS must be customized according to different purposes and peculiarities, we believe that it is possible to express extensions over interactions of software agents. In this paper, we will address the problem of how to prepare a law for extensions and how to refine law specifications. For this purpose, we enhanced the XMLaw description language [15] with some refinement operators to provide extensibility support and to realize a mapping of a customizable specification to the monitoring of interaction rules of governance mechanisms.

The main contribution of this proposal is to provide extensibility support within the interaction specification and compliance verification in open systems applications. For example, the operators allow the extension of the interaction laws including new services to run during the interaction monitoring and with filters to validate or not a message or norm.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ACM Reference Format
http://doi.acm.org/10.1145/1-59593-085-X/06/0005…$5.00.
Finally, the improvements in XMLaw were mapped to a law-governed mechanism that interprets these descriptions, plus their specializations, and analyzes the compliance of software agents that inhabit open software systems.

The organization of this paper is as follows. Section 2 describes an example of an open MAS that has evolved over the last years and it is used to describe our proposal. Section 3 details the law-governed approach, its architecture and some elements of the XMLaw description language [15]. In Section 4, we discuss variations in open MAS interactions and we describe how we included refinement operators in XMLaw. Section 4 also explains some examples of extension points identified in the example and we show how XMLaw can be used to support extensibility in a compliance mechanism. Related work is described in Section 5. Finally, we evaluate this approach and describe some future work and our conclusions in Section 6.

2. TAC SCM - An Example of Open MAS
A proof of concept prototype has been developed based on the specification of the Trading Agent Competition - Supply Chain Management (TAC SCM). TAC SCM [3][6][17] editions provide some evidence that the interaction specification evolves over time and so an extension support can reduce maintenance efforts.

The TAC SCM [3] has been designed with a simple set of rules to capture the complexity of a dynamic supply chain. SCM applications are extremely dynamic and involve an important number of information and resources among their different stages. In our case study, we mapped the requirements of TAC SCM into interaction laws and agents are implemented with JADE [4].

In TAC SCM, we chose the scenario of negotiation between the suppliers and assemblers to explain how extensions to interaction laws are used (Figure 2). According to [6], the negotiation process involves an assembler agent that buys components from suppliers. A bank agent also participates in this negotiation because an assembler may send RFQs to each supplier every day to order components offered by the supplier. Each RFQ represents a request for a specified quantity of a particular component type to be delivered on a specific date in the future. The supplier collects all RFQs received during the “day” and processes them. On the following “day”, the supplier sends back to each agent an offer for each RFQ, containing the price, adjusted quantity, and due date. If the agent wishes to accept an offer, it must confirm it by issuing an order to the supplier.

![Figure 1 Copy and Paste problem exemplified in XMLaw](image)

![Figure 2 - Roles, relationships and cardinalities of TAC SCM](image)

3. Governing Interactions in Open Multi-Agent Systems
Law-governed architectures can be designed to guarantee that the specifications of open systems will be obeyed. We developed an infrastructure that includes a communication component that is provided to agent developers [16]. This infrastructure, whenever necessary, permits extending this basic infrastructure to fulfill open system requirements or interoperability concerns. This architecture is based on a pool of mediators that intercept messages and interpret the previously described laws. Mediators monitor the conversations between components. As more clients are added to the system, additional mediators’ instances can be added to improve throughput.

In this section, we explain the XMLaw description language [15]. Here, XMLaw is used to represent the interaction rules of an open system specification. These rules are interpreted by a mechanism that at runtime analyzes the compliance of software agents with interaction laws [16]. Distributed software agents are independently implemented, i.e., the development is done without a centralized control. We assume that every agent developer may have a priori access to the open system specification, including the protocol descriptions and the interaction laws.

XMLaw represents the structure and the relationships of important law elements (Figure 3). 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.
In this section, we describe how norms are used to enhance scene and transition definitions; how constraints in norms and transitions act as filters of events; and how actions are used as an adaptation mechanism to support an active behavior of the environment in an open system. For further details about each of the concepts appearing in the conceptual model, please refer to [15]. Below, we will discuss XMLaw structure using the specification of laws for the TAC SCM example to facilitate its understanding.

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 components in an open system. Norms prescribe how the active distributed software components ought to behave, and specify how they are permitted to behave and what their rights 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.

### 3.1 Norms

There are three types of norms 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.

In TAC SCM, one permission norm was created about the maximum number of requests for quotation that an assembler can submit to a supplier. According to the TAC SCM specification [6], each day each agent may send up to a maximum number of RFQs. Besides this permission, the constraint on the acceptable due date of an RFQ regulates the same interaction, the request for quote message.

The structure of the Permission (Listing 1), Obligation and Prohibition elements are equal. Each type of norm contains activation and deactivation conditions. In the 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). Furthermore, norms define the agent role that owns it through the attribute Owner. In that case, the assembler agent will receive the permission. 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. Norms also have constraints and actions associated with them, but these elements will be explained later. Norms also generate activation and deactivation events that can be used by other law element.

---

**Listing 1: Permission Structure**

```xml
<Permission id="AssemblerPermissionRFQ">  
  <Owner>Assembler</Owner>  
  <Activations>  
    <Element ref="negotiation" event-type="scene_activation"/>  
  </Activations>  
  <Deactivations>  
    <Element ref="orderTransition" event-type="transition_activation"/>  
  </Deactivations>  
</Permission>
```

**Listing 2: Constraint in Transition Tags**

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

---

**3.2 Constraints**

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. A message pattern enforces the message structure fields [15]. However, 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. The tag `from` indicates the initial state of a transition and the tag `to` indicated the end state of a transition.
TAC rules; 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.

### 3.3 Actions

Environment actions, or just actions, are domain-specific Java codes that run integrated with XMLaw specifications. Actions can be used to plug services in an environment. For instance, an environment can call a debit service from a bank agent to automatically charge the purchase of an item during a negotiation.

Since actions are also XMLaw elements, they can be activated by any event such as transition activation, norm activation and even action activation. An action can be activated by as many events as are desired. The action structure is showed in the example of Listing 1. 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. In this example, the action is used to update the context of the norm, counting the number of submitted messages.

An action can be defined in three different scopes: Organization, Scene and Norms. An action defined in a Norm is only visible at this level. This means that any element in this scope can reference events issued by this action and that this action can get and update information at this level and upper levels. Actions defined in the scene scope can be referenced by any element at this level. And actions defined in the organization scope are visible to all elements at this level.

### 4. Refinement Operators to Specify Laws in Open Multi-Agent Systems

The definition of how the agents interact is very important to understand the open MAS behavior. The interaction specification is used as a guideline to enforce the expected behavior of agents in open MAS. Sometimes, the interaction laws that enforce the relationships between agents are not always fully understood early in the open MAS life cycle. Still, many more interaction laws are not applied because of a lack of systems support for changing interaction laws (i.e. extensibility) or because the interaction laws are exceptionally complex.

We argue that the interaction laws of open MAS should also be specified and developed to facilitate extensions to deal with this challenge. In this sense, it is necessary to have a way to specify which law elements can be customized and so defined as extension points. The extension points are 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 and actions into a pre-defined law specification. Even with extension points, the semi-complete law element specification can be referenced by other law elements.

The first attempt to define extension points was to defer the definition of the class implementation [5]; that is, the class implementation of constraints and actions was not defined (Listing 3). Since merely deferring the specification of class implementations does not clearly define when an element is an abstract element, the designer had to discover where the extensions were by browsing actions and constraints that did not reference a concrete class implementation.

```xml
Listing 3: Action and Constraint hook
```

The subsections below explain how the interaction specification with extension points can be better prepared for further refinements. We propose three operators to facilitate refinements in XMLaw specification: abstract, completes and extends.

#### 4.1 Identifying Extension Points: abstract

The documentation of an extension point is very important. The refinement operators must help the designer to find out where the semi-complete specifications are. An abstract operator can define law elements with extension points to be filled further on. It is also a means to clearly define the context where the extensions are expected. We still can defer the definition of the implementation of actions and constraints classes, as well as defining other law elements as abstract; and as we will see, we also can extend their definition by including new or superposing elements.

The abstract attribute defines when a law element is not completely implemented; it is useful 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").

In TAC SCM, the constraint checkDueDate (Listing 4) is associated with the transition rfgTransition. It means that if the verification is not true the transition will not be fired. The decision regarding the implementation of the checkDueDate constraint is deferred and so no class is specified in Listing 4.

```xml
Listing 4: Permission and Constraint over RFQ message
```

In Listing 5, there are two extension points: the constraint checkCounter and the action orderID. To customize this constraint and this action, we need to plug-in the class implementation. The constraint checkCounter is associated with the permission AssemblerPermissionRFQ. It means that if the verification is not true, the norm will not be valid, even if it is activated. The action ZeroCounter is defined under the permission AssemblerPermissionRFQ and it is triggered by a clock-tick every day, turning to zero the value of the counter of the number of requests issued by the assembler on this day. We do not provide further details regarding the clock definition. The other...
...action orderID is activated by every transition rfqTransition. It is used to count the number of RFQs issued by the assembler, updating a local variable.

```
<Permission id="AssemblerPermissionRFQ"
  abstract="true">
  <Owner>Assembler</Owner>
  <Activations>
    <Element ref="negotiation"
      event-type="scene_activation"/>
  </Activations>
  <Deactivations>
    <Element ref="orderTransition"
      event-type="transition_activation"/>
  </Deactivations>
  <Constraints>
    <Constraint id="checkCounter"/>
  </Constraints>
</Permission>
```

Listing 5: AssemblerPermissionRFQ Norm description

Another example of extension is given in the specification of the relationship between orders and offers of the negotiation protocol. According to Collins et al. [6], 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 6), 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 7).

```
<Permission id="ObligationToPay"
  abstract="true">
  <Owner>Assembler</Owner>
  <Activations>
    <Element ref="orderTransition"
      event-type="transition_activation"/>
  </Activations>
  <Deactivations>
    <Element ref="payingTransition"
      event-type="transition_activation"/>
  </Deactivations>
</Permission>
```

Listing 6: Obligation to pay specification

```
<Permission id="APRFQ2005"
  completes="AssemblerPermissionRFQ">
  <Constraint id="checkCounter"
    class="tacsdm.norm.constraints.CounterLimit2005"/>
  <Action id="orderID"
    class="tacsdm.norm.actions.RFQCounter2005"/>
</Permission>
```

Listing 8: Permission AssemblerPermissionRFQ extension

```
<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>
```

Listing 7: ObligationToPay usage

4.2 Filling the Gaps: completes

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 completes attribute is an operator that is useful to fill the elements that were left unspecified when a law element was defined as abstract. It is a simple operator to realize extensions since it can just be used to define action and constraints class implementations. The completes operator changes an abstract element into a concrete one and cannot leave any element unspecified. The completes operator cannot include any new element to the abstract law; it is limited to the definition of class implementations. For this specific purpose, we will present another operator, called extends (See Section 4.3).

Below, we present the refinements proposed to the law described above. In TAC SCM 2005 [6], each day each agent may send up to five RFQs to each supplier for each of the products offered by that supplier, for a total of ten RFQs per supplier. For this refinement, another action component named RFQCounter2005 is plugged-in (Listing 8). It counts the number of RFQs according to the type of component. The constraint CounterLimit2005 was also chosen as a specific counter for each type of component that a supplier provides.

```
<Transition id="rfq2005" completes="AssemblerPermissionRFQ">
  <Constraint id="checkDueDate"
    class="tacsdm.constraints.ValidDate2005"/>
  <Action id="orderID"
    class="tacsdm.norm.actions.RFQCounter2005"/>
</Transition>
```

Listing 9: Constraint checkDueDate extension

4.3 OO Specialization: extends

The supplier will not consider RFQs with due dates beyond the end of the game, or with due dates earlier than two days in the future, will not be considered. This requirement is implemented by the constraint ValidDate2005 (Listing 9). Notice that if we want to extend this law to other editions of TAC SCM, we just need to define and associate new implementations of these actions and constraints.

```
<Transition id="rfq2005" completes="rfqTransition">
  <Constraint id="checkDueDate"
    class="tacsdm.constraints.ValidDate2005"/>
</Transition>
```

Listing 10: Constraint checkDueDate extension

According to [6], suppliers wishing perhaps to protect themselves from defaults will bill agents immediately for a portion down of
the cost of each order placed. The remainder of the value of the order will be billed when the order is shipped. In TAC SCM 2005, the down payment ratio is 10%. This down payment is implemented by the action SupplierPayment (Listing 10). Notice that we have added a definition regarding the existence of an action in the context of the obligation definition.

```xml
Listing 10: ObligationToPay extension for TAC SCM 2005

4.4 Interpreting Refinement Operators and Laws

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. Also, the completes operator cannot include any new element to the abstract law; it is limited to the definition of class implementations. Finally, the completes and extends operator turns an abstract element into a complete one and cannot leave any element unspecified.

4.5 Law Execution Order after Extensions

In XMLaw, the composition and interrelationship among law elements is done by events. Every law element is related to events; one law element can generate events to signal something to other elements. Other elements can sense events for many purposes, for instance, activating or deactivating themselves. In this sense, the order of execution must be considered and clearly understood even with extensions; i.e., we need to clearly define the order of elements’ activation in the case of using the refinement operations.

The event monitoring model is implemented using the observer pattern [9]. During law interpretation, new elements are attached to the observer structure. Laws are interpreted in at least two steps: first, the base law is read and then extensions are attached to the execution model. Any new element defined after an extension is associated with the execution model after the basic elements and then their activation will occur after the activation of basic elements.

Let us consider the example scenario (Figure 4) where there is a hierarchy of extensions for a payment policy that consider the client’s importance for the bank. The discount action will calculate the percentage of discount that a client can have on his payment (it is possible to accumulate discounts). During system execution, suppose that the monitoring mechanism receives an event PAYMENT from a PRIME Client, and then the expected order of activation of actions will be discount 10%, then discount 5% and finally discount 20%. If the product cost is 100 units, at the end of this execution it would cost 68.4 units.

Figure 4 - Example of the execution order of law elements

5. Related Work

We do not have enough space in this paper to fully describe all aspects regarding software evolution in general or evolution in XML-based languages. However, we chose to detail some related work that are directly related with evolution of interaction laws and interaction protocol specifications. We included in XMLaw one mechanism for extensions defined a priori, namely completes, and another for open extensions, extends, analog to inheritance, like often in programming languages.

Minsky [2][14] proposes a coordination and control mechanism called law-governed interaction (LGI). Ao and Minsky [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. In contrast, we focus on interaction extensibility support. 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 [10] 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 [12], 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 [11] 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 [19] 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 [19] has its focus on building multi-agent systems instead of monitoring and enforcement purpose.

In Esteva’s [7] approach, scenes and protocol elements specify the interaction protocol using a global view of the interaction. XMLaw includes the concept of actions and constraints, which
allows execution of Java code in response to some interaction situation, and we use them to implement the extension points.

6. Conclusions and Future Work
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 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.

While analyzing the open software system domain, it is possible to distinguish two groups of specifications concerning an agent’s interactions: fixed (stable) and flexible (extensible). By this analysis, it is possible to design part of the open system evolution in the solution. If a desired characteristic of a system is long-term stability, then the challenge to developers is to deliver a product that identifies the aspects of the open MAS that will not change and cater the software to those areas. Besides some basic services, in open systems, system stability is characterized by the interaction protocol and some general rules that are common to all open MAS instances. Extensions of interaction rules will impact the open MAS and the agents and extensions are specified. It is our interest to continue to research these topics, so we will continue to enhance XMLaw to support interaction extensibility specification.

We are aware of possible problems about consistency 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.

7. Acknowledgments
We gratefully acknowledge the financial support provided by the CNPq as part of individual grants and of the ESSMA project (552068/2002-0) and by CAPES as part of the EMACA Project (CAPES/COFECUB 482/05 PP 016/04).

8. References