PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 03/07

Event-Driven High Level Specification of Laws in Open Multi-Agent Systems

Rodrigo de Barros Paes
Carlos José Pereira de Lucena
Gustavo Robichez de Carvalho
Don Cowan

Departamento de Informática

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

Monografias em Ciência da Computação, No. 03/07 ISSN: 0103-9741
Editor: Prof. Carlos José Pereira de Lucena February, 2007

Event-Driven High Level Model Specification of Laws in Open Multi-Agent Systems *

Rodrigo de Barros Paes, Carlos José Pereira de Lucena, Gustavo Robichez de Carvalho, Don Cowan[1]

{rbp,lucena,guga}@inf.puc-rio.br

[1]University of Waterloo

dcowan@csg.uwaterloo.ca

Abstract. The agent development paradigm poses many challenges to software engineering researchers, particularly when the systems are distributed and open. They have little or no control over the actions that agents can perform. Laws are restrictions imposed by a control mechanism to deal with uncertainty and to promote open system dependability. In this paper, we present a high-level event driven conceptual model of laws. XMLaw is an alternative approach to specifying laws in open multi-agent systems that presents high level abstractions and a flexible underlying event-based model. Thus XMLaw allows for flexible composition of the elements from its conceptual model and is flexible enough to accept new elements.

Keywords: Governance, Agents, Protocols, Electronic Institutions, Multi-Agent Systems, Open Systems.

Palavras-chave: Governança, Agentes, Protocolos, Instituições Eletrônicas, Sistemas Multi-Agentes, Sistemas Abertos.

---

* Trabalho parcialmente patrocinado pelo Ministério de Ciência e Tecnologia da Presidência da República Federativa do Brasil (e agência de fomento e o número do processo, se aplicável). (Em Inglês: This work has been sponsored by the Ministério de Ciência e Tecnologia da Presidência da República Federativa do Brasil)

Responsável por publicações (ou In charge for publications, se o texto for em inglês):

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

1 Introduction

The agent development paradigm poses many challenges to software engineering researchers, particularly when the systems are distributed and open to accepting new modules that have been independently developed by third parties. Such systems have little or no control over the actions that agents can perform. As open distributed applications proliferate the need for dependable operation becomes essential.
There has been considerable research addressing the notion that the specification of such open multi-agent systems (MAS) should include laws that define behaviors in an open system [17][18][19]. Laws are restrictions imposed by a control mechanism to tame uncertainty and to promote open system dependability [6][20]. In this sense, such a mechanism should perform an active role in monitoring and verifying whether the behavior of agents is in conformance with the laws. We call this mechanism a governance mechanism. Examples of governance mechanisms are LGI [6], Islander [10] and MLaw [21].

Governance for open multi-agent systems can be viewed as an approach that aims to establish and enforce some structure, set of norms or conventions to articulate or restrain interactions in order to make agents more effective in attaining their goals or more predictable [22].

A governance approach has to deal with two important issues: a conceptual model (also called a domain language, or meta model) and the implementation mechanism that supports the specification and enforcement of laws based on the conceptual model. The content of this paper is mainly about the former.

In the conceptual model, the approach describes what elements designers can use when specifying the law. The model specifies the vocabulary and the grammar (or rules) that designers can use to design and implement the laws. The model has a decisive impact on how easy it is to specify and maintain the laws. It is the approach to design that largely determines the complexity of the related software. When the software becomes too complex, the software can no longer be well enough understood to be easily changed or extended. By contrast, a good design can make opportunities out of those complex features [23].

In 1987, Minsky published the first ideas about laws [27] and in 2000, he published a seminal paper about the role of interaction laws on distributed systems [6], which he called Law-Governed Interaction (LGI). Since then he has conducted further work and experimentation based on those ideas [24][25][26]. Although LGI can be used in a variety of application domains, its conceptual model is composed of abstractions basically related to low level information about communication issues (such as the primitives disconnected, reconnected, forward, and sending or receiving of messages). While it can be possible to specify complex interaction rules based on such low level abstractions, they may not be adequate for the design of the laws pertaining to complex systems. This inadequacy occurs because, once the laws in the domain level are mapped to the many low level primitives, the original idea of the law is lost, as it is spread over many low level primitives. This is basically the problem of having a language that provides abstractions that are too far removed from the domain. When developing complex interactive systems, we need higher-level abstractions to represent laws in order to reduce complexity and achieve resultant productivity.

The Electronic Institution (EI) [10] is another approach that provides support for interaction laws. An EI has a set of high-level abstractions that allow for the specification of laws using concepts such as agent roles, norms and scenes. Historically, the first ideas appeared when the authors analyzed the fish market domain [28]. They realized that to achieve a certain degree of regulation over the actions of the agents, real world institutions are needed to define a set of behavioral rules, a set of workers (or staff agents), and a set of observers (or governors) that monitor and enforce the rules. Based on these ideas, they proposed a set of abstractions and a software implementation. However, although EI provides high level abstractions, its model is quite inflexible with respect to change. The property of flexibility is quite important since the research in interaction laws is under constant evolution, and consequently the model that represents the law abstractions and their underlying implementation should also be able to evolve. One example of evolution is the use of laws for providing support to the implementation of dependability concerns. In this situation, the monitoring of laws may allow the detection of unexpected behaviors of the system and corresponding recovery actions.

The question is how can a conceptual model of laws evolve if we do not know in advance the nature of the changes? The answer to this question is that there is no way to foresee which parts are going to change. However, it is possible to use a basic underlying model that is inherently flexible. Event-based systems lead to flexible systems mainly because they avoid direct dependencies among the modules. Instead, the dependency is between the modules and the events they produce or consume.

In general, event-driven software design avoids any direct connection between the unit in charge of executing an operation and those in charge of deciding when to execute it. Event-driven techniques lead to a low coupling among modules and have gained acceptance because of their help in building flexible system designs [1]. In an event-based architecture, software components interact by generating and consuming events. When an event in a component (called source) occurs, all other components (called recipients) that have declared interest in the event are notified. This paradigm appears to support a flexible and effective interaction among highly reconfigurable software components [2], and has been applied successfully in very different domains, such as graphical user interfaces, complex distributed systems [2], component-based systems [4] and software integration [5]. Many of these approaches use event-based systems to manage changes in the software that cannot be anticipated during design [4][3]. Such changes are generally driven by a better understanding of the domain, and by external factors (such as strategic, political or budget decisions).

In this paper, we present a high-level event driven conceptual model of laws. The focus is to highlight the "high-level" and "event-driven" aspects of the model, instead of presenting in detail the model itself. We do not claim that the abstractions of the proposed conceptual model are better than the ones in related approaches. Instead, we claim that the model is composed of a rich set of high level abstractions which enable, for instance, the specification of complex laws that can even interact with many current technologies (such as web services). The model is specified based on the event-driven paradigm. As a result, new elements can be easily introduced in the model.

The idea is that each element should be able to listen to and generate events. For example, if the model has the notion of norms, then this norm element should generate events that are potentially important to other elements, such as lifecycle events, norm activation, sanction applications and so on. The norm is also able to listen for events generated by other elements of the conceptual model, and then can react accordingly. For example, if in the conceptual model there is an element that models the notion of time, such as an alarm clock, then norms may listen to alarm clock notifications and their behavior becomes sensitive to time variations. This leads to very flexible and powerful relationships among the elements. Furthermore, if there is a need to introduce a new element in the model, then most of the work is restricted to connecting this new element to the events to which it needs to react, and to discover which events this new element should propagate.

The flexibility achieved by using the event-driven approach at a high-level of abstraction is not present in the other high-level approaches [10][34]. The advantages claimed by the use of events as a modeling element are also present in Minsky's approach [6] as a low level of abstraction. In this paper, we show in detail how to map our high level approach to Minsky's in such a way that we illustrate that we can also achieve all the results he has produced so far (we are not addressing efficiency and security issues).

At the implementation level, we have developed middleware that supports the interpretation and enforcement of the specification and treats each element of the conceptual model as a component that is able to generate and sense events. The presentation of the middleware implementation is outside the scope of this paper and can be found in [15][13].

This paper is organized as follows. Section 2 shows the proposed solution as a step in solving the stated problem. In section 3 we relate our research to previous work, explaining how the problem of flexibility and evolution has been addressed. Section 4 shows two case studies where the model is applied and compared to related work. Finally, in Section 5, we present some discussions about the contents of this paper and future work.

2 XMLaw: An Event-Driven Model

In this section, we present a partial view of XMLaw the conceptual model of laws and show how a new element can be added and connected to other elements through events. The full description of the XMLaw is described in [11] [20].

Fig. 1 shows the elements that compose the XMLaw conceptual model. The term conceptual model has the same meaning adopted by OMG[1] to refer to the UML conceptual model. This model can be viewed as composed of elements that cover many dimensions of the design of laws. As we could not find any related taxonomy in the literature, we are using the following ad-hoc classification:

Time – this dimension supports the specification of laws that are sensitive to time. For example, certain rules can have deadlines, expiration dates, and cyclical time-dependent behavior.

In the model, the Clock element represents this dimension. Clocks represent time restrictions or controls and 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.

Social – this dimension supports the social relationships and interactions among the agents. Examples of social relationships are master-slave in some distributed systems and employer-employee in company environments.

1 Object Management Group – www.omg.org

The model has three elements in this dimension: Agent, Role and Message. The Agent represents a software agent that is interacting with the other agents under the rules of the laws. There is no assumption about what language or architecture has been used to implement the agents. A role is a domain-specific representation of the responsibilities, abilities and expected behavior of an agent. It is useful to provide an abstraction for roles that is not related to the individuals playing the role. The Message element models a message exchanged among agents.

Structural – this dimension encompasses every type of structure used to describe the laws. They usually define modular contexts for which the laws are valid.

There are two elements in this dimension: Scene and Law. The idea of scenes is similar to the one in theater plays, where actors follow well defined scripts, and the whole play is composed of many sequentially connected scenes. A scene models a context of interaction where a protocol, actions, clocks and norms can be composed to represent complex normative situations. Furthermore, from the problem modeling point of view, a scene allows decomposition of the problem into smaller and more manageable pieces of information. The Law element is a context where all the other elements can be grouped. This is the most general element, and it does not belong to any other element.

Restrictive – this dimension contains elements with a focus on restricting the set of actions that agents are allowed to perform in a given context. The model provides five elements in this dimension: Protocol, State, Transition, Norm, and Constraint.

A protocol defines the possible states through which an agent interaction can evolve. Transitions between states can be fired by any XMLaw event. Therefore, protocols specify the expected sequence of events in the interactions among the agents.

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 obliged 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. For example, the winner of an auction has permission to interact with a bank provider through a payment protocol. Finally, the prohibition norm defines forbidden actions for a software agent at a given moment; for instance, if an agent does not pay its debts, future participation in a scene will be prohibited.

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. 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 a constraint as it is needed for their applications.

Service – this dimension is related to the interaction between the laws and the services that exist in the environment.

The action element belongs to this dimension. An action supports the definition of the moment when the mediator should call a domain-specific service.

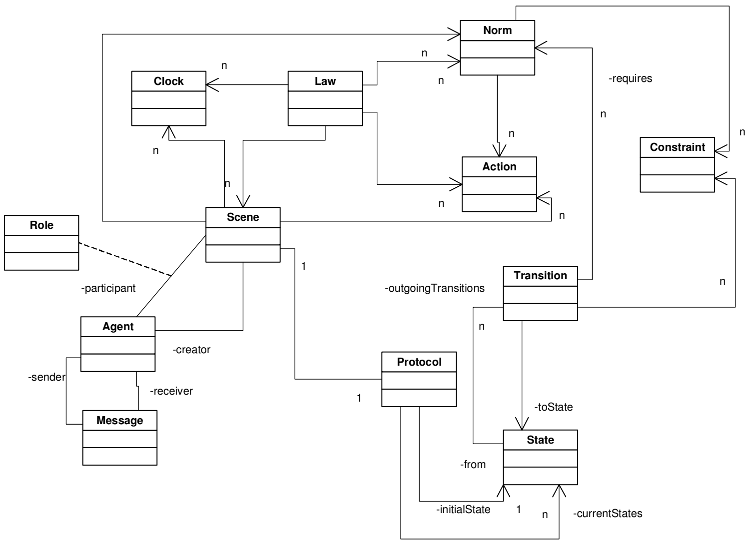

Fig. 1. The XMLaw conceptual model

Those elements of the model are connected through an event-based paradigm making it possible to achieve flexible behaviors through composition. For example, the norm element can be composed with the transition that creates an awareness of transition activation and thus behave properly. The pseudo code listed in Table 1 illustrates this idea. Other elements can also be composed to achieve complex behavior. For example, suppose that the buyer from the norm 1 in Table 1 now has to fulfill the obligation in at most 10 minutes. Then, this norm should incorporate some sense of time. In order to achieve that, we could compose the norm with the clock to reach the desired effect. Table 2 shows how this composition could be achieved. First, the clock listens to hear when the norm is given to the buyer. Then, the clock starts to count the ten minutes, and when this time has elapsed, the clock generates a *clock_tick* event. This *clock_tick* is then heard by norm2, which then prohibits any further interaction for the agent playing the buyer role.

In this scenario, the norm element was not originally designed or conceived to incorporate the notion of time, and the clock did not incorporate the notion of norms. The low coupling among these two elements caused by an event-based approach has lead to a flexible model of composition, even when the composition is not anticipated.

Table 1 – Pseudo code for activating a norm due to the firing of a transition

```
…
t1(s0,s1, message_arrival(m1) )
…
norm1{
   give obligation to buyer when listen(transition_activation(t1))
}
…
```

Table 2 – Pseudo code for composing the norm with the clock

```
…
t1(s0,s1, message_arrival(m1) )
clock1{
   start to count when listen(norm_activation(norm1))
   count until 10 min and generate(clock_tick(clock1))
}
…
norm1{
   give obligation to buyer when listen(transition_activation(t1))
}
norm2{
   prohibit all interactions from buyer when listen(clock_tick(clock1))
}
…
```

The evolution of the conceptual model of XMLaw has been influenced by the experiments we have been conducting. One special evolution applied to the original model was driven by the need for interacting with some services provided by the environment. In some cases, the laws could specify when and how to perform recovery actions; notify other stakeholders about changes in the law (through web services, for example), update database information and so on.

Based on these experiences, we have specified one more element, which is called Action. Actions are domain-specific Java code that runs in an integrated manner with XMLaw specifications. Actions can be used to plug services into a governance mechanism. For instance, a mechanism can call a debit service from a bank agent to charge the purchase of an item automatically during a negotiation. In this case, we specify in the XMLaw that there is a class that is able to perform the debit. Of course, this notion could also be extended to support other technologies instead of Java, such as direct invocation of web-services. Thus, these experiments demonstrate how a new action element can be included in the event-based model.

Within the event model, the action can be integrated with the other elements by making actions able to listen to the other events. In this way, it would be possible to activate an action because of a clock activation, a norm activation, transitions and all the other events. On the other hand, to make the other elements able to react to actions, there is no need for changes, once the other elements can sense events; one can specify a listener for action activations.

Table 3 shows a situation where the action is activated by a transition, and then a clock is activated because of the action.

Table 3 – Pseudo code for composing the norm with the clock

```
…
t1(s0,s1, message_arrival(m1) )
clock1{
   start to count when listen(action_activation(action1))
   count until 5 sec. and generate(clock_tick(clock1))
}
action1{
   run "charge item in the bank" when listen(transition_activation(t1))
}
…
```

Although, the examples shown in this section are simple, they illustrate the consequences of having a flexible conceptual model. The conceptual model is usually mapped to some language (graphical or textual) that allows the specification of laws. The second step is to build an interpreter that is able to read the specification and verify the compliance of the specification with the actual system behavior. The underlying event-based model can also be smoothly mapped to the implementation level, so the interpreter will also be flexible as it follows the same principles. We have implemented middleware, called M-Law [15] (Middleware for LAWs), that uses a component-based abstraction to represent each element of the conceptual model and an event-based model to make communication among the components possible.

2.1 The Event-Driven Model Definition

All the elements of the meta-model are able to sense and generate events, more precisely, let E be the set composed of the following elements: {Law, Scene, Norm, Clock, Protocol, State, Transition, Action, Constraint, Agent, Message, Role}, and let e denote an element of E, i.e., $e \in E$, then each e is able to sense and generate events.

As can be seen in Fig. 2, every e has the same basic lifecycle. It is important to notice that there is no restriction over which event can activate an element, this information provided through the law specification and therefore it is loosely coupled with the model.

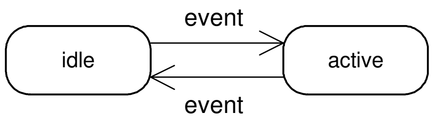

Fig. 2. Generic Law Element Lifecycle

Table 4 shows an example where the specification of the type of event that activates an element, in this case a clock, is just expressed in the law (for readability purposes the codes written in XMLaw presented in this paper use a simplified syntax which is more compact than the one used in early XMLaw publications). Line 16 says that a clock is activated (goes from idle to active state) when transitions t1 or t4 fire, and it is deactivated (goes from active to idle state) when transitions t2, t3 or t4 fire.

Table 4 – Defining the events that activate an element

```
…
08:    t1{s1->s2, propose}
09:    t2{s2->s3, accept}
10:    t3{s2->s4, decline}
11:    t4{s2->s2, propose}
...
16:    clock{5000,regular, (t1,t4),(t2,t3,t4)}
…
```

This very simple event mechanism has important consequences. It allows for a flexible and uncoupled composition of elements, and also allows for changes in the model. For example, when it is necessary to handle dependability concerns [30][31].

3 Relating the Model to Other Approaches

3.1 Relating the Model to a Lower Level Event-Based Approach

Minsky [6] proposed 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 a home agent is the agent being regulated by the laws; the ruling for an event *e* can depend only on *e* itself, and on the local home agent's context; and the ruling for an event can mandate only local operations to be carried out at the home agent. On the other hand, the decentralization of law enforcement is an architectural decision argued as necessary for achieving scalability.

LGI has a rich set of events that can be monitored on each controller. Once these events are monitored, it is possible to use operations in order to implement the law. The union of events and operations is the conceptual model of LGI. The LGI conceptual model was conceived to deal with architectural decisions to achieve a high degree of robustness. This has lead to a model composed of low level primitives. Although the primitives are adequate for many classes of problems, it is necessary sometimes to use various primitives to achieve the desired effect. Once the laws become larger and more complex, it can be hard to maintain such a set of low level primitives.

One can think of LGI as a highly scalable virtual machine whose instructions are made of low level law elements. In this way, it would be possible, for example, to use high level abstractions of XMLaw to specify the laws, and in a second step map the specification to run on top of the LGI architecture. In order to illustrate this idea, we show how some of the elements that compose the conceptual model of XMLaw can be mapped to several of the LGI primitives. The illustration can easily be extended to cover all elements of the XMLaw model and the LGI architecture. We have summarized the main regulated events and operations of LGI in Table 5 and

Table **6**.

Table 5 - Main regulated events of LGI approach. (This list is not intended to be complete.)

Regulated Events	
adopted	Represents the birth of an LGI agent – more specifically, this event represents the point in time when an actor adopts a given law L under which to operate thus becoming an L-agent.
arrived	This event occurs when a message M sent by agent X to agent Y, arrives at the controller of Y. (The home of this event is agent Y – the receiver.)
disconnected	This event occurs at the private controller of an agent when its actor has been disconnected.
exception	This event may occur when the primitive operation, which has been invoked by the home agent, fails.
obligationDue	This event is analogous to the sounding of an alarm clock, reminding the controller that a previously imposed obligation of a specified type is coming due. Obligations are imposed by means of the primitive operation imposeObligation,
reconnected	This event occurs at the private controller of an agent when its previously disconnected actor has been reconnected.
sent	This event occurs when the actor of x sends a message m addressed to an agent y operating under law L'. The sender x is the home of this event.
stateChanged	This event occurs at an agent x when a pending state-obligation at x comes due.
Submitted	This event, which is a counterpart of the arrived event, occurs at an agent x, when an unregulated message m sent by some process at host h, using port p, arrives at x. It is, of course, up to law L under which x operates to determine the disposition of this message.

Table 6 - Main regulated operations of LGI approach. (This list is not intended to be complete.)

Operations	
Deliver	This operation, which has the form deliver([x,L'],m,y), delivers to the home actor the message m, ostensibly sent by x, operating under law L'
Forward	Operation forward(x,m,[y,L']) sends the message m to Ty, the controller of the destination y—assummed here to operate under law L'; x is identified here as the ostensible sender of this message.
Add	adds term t to the CS.
Remove	Removes from CS a term that matches t, if any. If there is no such term to be removed, this operation has no effect.
Replace	Replaces a term t1 from CS, if any, with term t2. If there is no term t1 to be replaced, then this operation has no effect.
Incr	Operation incr(f,d), locates a unary term f(n), and increments its argument by d.
Decr	decr(f,d) is the implied counterpart of incr.
replaceCS	Operation replaceCS(termList) replaces the whole control-state of the home agent with the specified list of terms.
addCS	Operation addCS(termList) appends the terms in the list termList to the control state of the home agent.
imposeObligation	imposeObligation(oType, dt, timeUnit) imposes an obligation of the specified type on the home agent, to come due after a delay dt, given in the specified time units.
repealObligation	Operation repealObligation(oType)removes all pending obligations of type oType, along with all associated obligation-terms in DCS.
imposeStateObligation	Operation imposeStateObligation(termList) would cause a state-Changed) event to occur upon any change in any of the terms of the CS that are indicated by the termList parameter.
repealStateObligation	Operation repealStateObligation(all) repeals the current state-obligation, as well as the corresponding audited(termList) term from the DCS

Most of the events found in Table 5 are related to low level information about communication issues (*disconnected, reconnected*), sending or receiving of messages (*sent, arrived*), or state changes on the control state (*stateChanged*). From the point of view of the operations, they are also mostly concerned with low level instructions such as *forward*, *deliver*, *add* and so on. XMLaw has a rich set of high level abstractions. In

Fig. 3 we show how to map some of abstractions to LGI instructions while preserving the meaning.

The protocol presented in Fig. 3 can be directly specified in XMLaw through the elements Protocol, State, Transition and Message. In order to achieve the same behavior in LGI, one can write the law as illustrated in Table 7. In this LGI law, we have introduced two terms: currentState and event. The term currentState models the current states of the protocol, and the term event simulates generation of events. For example, in the first line of Table 7, an agent A sends the message m1 to the agent B; if the current state is s0, then state s0 is removed from the list of current states, and state s1 is added to the list. We also simulate the generation of a *transition_activation* event, and finally the message is forwarded.

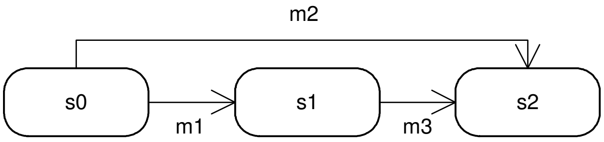

Fig. 3. Protocol Example

Table 7 - Protocol specification in LGI

```
sent(A,m1,B) -> currentState(s0)@CS, do(remove(currentState(s0))),
do(add(currentState(s1))), do(add(event(t1,transition_activation))), do(forward).
sent(A,m2,B) -> currentState(s0)@CS, do(remove(currentState(s0))),
do(add(currentState(s2))), do(add(event(t2,transition_activation))), do(forward).
sent(B,m3,A) -> currentState(s1)@CS, do(remove(currentState(s1))),
do(add(currentState(s2))), do(add(event(t3,transition_activation))), do(forward).
```

Further examples in Table 8 show how some situations found in XMLaw can be mapped to the LGI approach, by incorporating the terms *event, currentState,* and *norm* in the LGI semantics expressed in prolog.

Table 8 - XMLaw situation modeled using LGI

1. Upon the arrival of a message *m1*, a clock must be activated to fire an event in 5 seconds. The clock should be deactivated when the message *m2* arrives.
XMLaw myXMLawClock{5000,regular, (m1),(m2)}
LGI arrived(X, m1, Y) :- imposeObligation("myLGIclock",5). arrived(X, m2, Y) :- repealObligation("myLGIclock”).
2. Fire transition *t2* when the clock generates a *clock_tick* event. The transition changes the protocol from state *s1* to state *s2*.
XMLaw t2{s1->s2, myXMLawClock}

LGI
obligationDue("myLGIclock ") :- currentState(s1)@CS, do(remove(currentState(s1))),
do(add(currentState(s2))), do(add(event(t2,transition_activation))), do(forward).
// comments: the obligationDue event happens when the time specified in
// the obligation expires.

3. Declare a periodic clock that must generate an event each five seconds. This clock should be activated by the arrival of message *m1* and deactivated by the arrival of message *m2*.
XMLaw myXMLawClock{5000,periodic, (m1),(m2)}

LGI
arrived(X, m1, Y) :- imposeObligation("myLGIclock",5).
arrived(X, m2, Y) :- repealObligation("myLGIclock”).
obligationDue("myLGIclock") :- imposeObligation("myLGIclock ",5).
// comments: the obligationDue event happens when the time specified in
// the obligation expires. We can use a loop in the specification for //simulate periodic clocks. In this example, first we “declare” a clock, then //when this clock expires, then an obligationDue event is generated that in //its turn activates another imposeObligation and so on

3. Message *m1* activates transition *t1*. The transition *t1* changes the protocol state from *s1* to *s2*. The norm *n1* must be activated when transition *t1* is fired. The norm is given to the agent that received the message *m1*.
XMLaw t1{s1->s2, m1} n1{$addressee, (t1) }
LGI

```
sent(A,m1,Addresee) -> currentState(s1)@CS, do(remove(currentState(s1))),
do(add(currentState(s2))), do(add(event(t1,transition_activation))), do(forward).
imposeStateObligation( event(t1,transition_activation) ).
stateChanged(event (t1,transition_activation)) :- do( add( event(n1,norm_activation) )
), do(add(norm(n1,active,valid)).
// comments: In this case, the imposeStateObligation would cause an //stateChanged event
whenever the event(t1,transition_activation) term is //added to the CS. Then, the third
command states that when this //stateChanged happens, then the norm n1 is made active.
```

3.1.1 Global Properties

According to [12], "any policy that can be implemented via a central mediator—which can maintain the global interaction state of the entire community—can be implemented also via an LGI law". As an example of a global property, suppose we encounter the situation in **Fig. 4**. In this example, 3 agents are interacting in the context of a specified protocol. Agent A sends the message m1 to Agent B. As agents A and B interact, their controllers update the current state.. However, agent C has not participated of this interaction and therefore, its controller has not updated its current state. This causes an inconsistency between the agents A and B states and the state of agent C. This happens because the monitoring is performed in a decentralized way with no explicit synchronization.

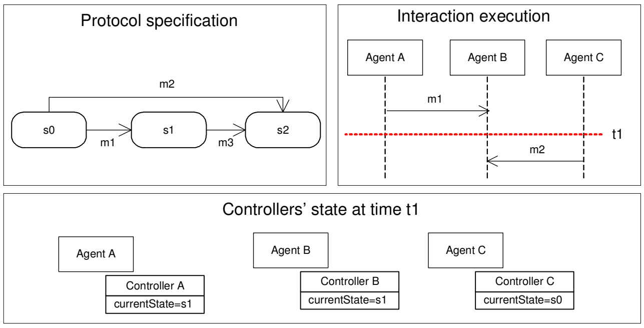

Fig. 4. Example of the need for synchronization of controllers to preserve global properties

The general way to overcome this problem is to have specific synchronization protocols such as the token ring in the Islander approach [10]. In LGI it could be achieved by introducing a central coordinator that receives all the messages, and therefore keeps a consistent global state. Table 9 outlines an implementation in LGI. The laws in XMLaw are specified from a global point of view [13]. The approach presented in Table 9 has one advantage over the centralized mediator used in XMLaw, since the XMLaw mediator cannot protect itself against congestion because of some overactive participants which may lead to denial of service. Under LGI, on the other hand, the law may limit the frequency of messages that can be issued by any given participant. This limit can be locally enforced, and is less susceptible to congestion [6].

Table 9 - Redirecting to a central coordinator

```
alias(coordinator,'law-coordinator@les.inf.puc-rio.br').

// any message is forward to the central coordinator
sent(X,M,Y) :- do(forward(X,[M,Y],#coordinator)), do(forward).

// laws ...and redirection to the real addressee
arrived(#coordinator,A,m1,B) -> currentState(s1)@CS, do(remove(currentstate(s1))),
do(add(currentState(s2))), do(deliver), do(forward(A,m1,B)).
```

3.2 Relating the Model to a Not Event-Based Approach

Electronic Institutions [10] are a technology to enforce and monitor the laws that apply to the agent society in a given environment. Several case studies were presented using this approach. They include a Fish Market system [7], a Grid Computing Environment application [8], and a Traffic Control application [9].

Electronic Institutions (EI) uses a set of concepts that have points of intersection with those used in XMLaw. For example, both EI scenes and protocol elements specify the interaction protocol using a global view of the interaction. The time aspect is represented in the Esteva's approach [10] as timeouts. Timeouts allow activating transitions after a given number of time units have passed since a state was reached. On the other hand, because of the event model, the clock element proposed in XMLaw can both activate and deactivate not only transitions, but also other clocks and norms. Connecting clocks to norms allows for a more expressive normative behavior; norms become time sensitive elements. Furthermore, XMLaw also includes the concept of actions, which allows execution of Java code in response to some interaction situations.

Table 10 compares the abstractions used in the conceptual model of both approaches. The goal of this comparison is to relate XMLaw better with an already existing well-known approach. The comparison shows that although they share a good set of concepts they have some important differences. For example, the notion of norms presented in EI [14] is better defined than in XMLaw. On the other hand XMLaw has the concept of Actions that can be useful for making the laws behave more actively and integrated with services provided by the environment. However the major difference between the two models is the way that abstractions are related to compose the law. In EI there is a fixed set of relationships among the elements, and the way elements are used together is already defined in advance. A good example of this is the timeout abstraction. Timeout abstraction of EI is very similar to the clock abstraction of XMLaw. However, one can only use the timeout with transitions. If the underlying communication model among the elements were more flexible, it would be possible for example, to do the same thing as XMLaw and connect the timeout to the norm.

Table 10 – Relating EI and XMLaw conceptual models

Electronic Institutions	XMLaw	Comments
Illocutory formulas	Message	They have different structures but mean the same.
EI vocabulary (ontology)	It is defined in the messages themselves, instead of separately.	EI defines an explicit ontology of all the terms used in the conversation. XMLaw does not require this definition.
Internal roles	Not considered	Internal roles define a set of roles that will be played by staff agents which correspond to employees in traditional institutions. Since an EI delegates their services and duties to the internal roles, an external agent is never allowed to play any of the roles.
External roles	Role	
Relationships over roles	Not considered	
Control over role playing	Control over role playing	Both approaches provide control over the minimum and maximum number of agents that can play a role in a scene.
Scene	Scene	Both approaches have the notion of scene. In EI it is necessary to specify which agents are allowed either to enter or to leave a scene at some particular moments. In XMLaw, there is no need to specify the exit moments. This is because as agents can fail, or even exit at their own will, XMLaw considers exit moments as not necessary.
Performative Structure	Not considered.	The Performative structure is a special type of scene that accepts transitions from other scenes and has outgoing transitions to other scenes. They allow for a specification in which sequence scenes are expected to happen. In XMLaw, there is no such concept; however the notion of norm can be used to achieve similar effect. Once in the end of a scene a norm could be activated and checked against the start of a new

		scene.
Protocol	Protocol	
State	State	States are quite similar, except that XMLaw has two types of final states: failure and success.
Directed edge	Transition .	The directed edge of EI (we used this name because EI has a transition element in the performative structure that has a different meaning) can be activated by illocution schemata, timeouts or constraints. In contrast, transitions in XMLaw can be activated by any event.
Constraint	Constraint	In EI, constraints are specified as boolean expressions using a operator and two expressions: (op expr1 expr2). In XMLaw, they are implemented as domain-dependent Java Code.
Time-out	Clock	Time-out allows provoking transitions after a given number of time units have passed since the state was reached. In contrast, clock is a general purpose clock that can also be used to provoke transitions to fire. But it can also be used, for example, to give an expiration period for a norm.
Normative rules	Norms	Both approaches model notions of obligations, permissions and prohibitions. Normative rules model the notion of obligation by verifying: "when an illocution is made and the illocution satisfies certain conditions THEN another illocution with other conditions must be satisfied in the future". The norm in XMLaw can be used to prevent transition activations, actions activations and so on
Not considered	Actions	Actions can be used to plug services in the mediator. They can be activated by any event such as transition activation, norm activation and even action activation. The action specifies the Java class in charge of the functionality implementation
Not considered	Law	In XMLaw, the Law element is a global context where shared information among scenes and norms, clocks and actions can be used. In EI, a closed effect can be achieved through the performative structure.

4 Case Studies

In this section, we have chosen two examples published in the literature to illustrate the applicability of the XMLaw model. The first example was already implemented and reported using the LGI approach. The second was also implemented and reported using the EI approach. By choosing these examples, we are able to compare the pros and cons of the various approaches directly.

4.1 Case Study 1: Buyer Team

This case study was already implemented with LGI and presented in [32]. There are some modifications to the original problem description. We have eliminated the need for a certification authority. The example is described as follows.

"Consider a department store that deploys a team of agents, whose purpose is to supply the store with the merchandise it needs. The team consists of a manager, and a set of employees (or the software agents representing them) who are authorized as buyers and have access to a purchasing-budget provided to them.

Let us suppose that under normal circumstances, the proper operation of this buying team would be ensured if all its members comply with the following, informally stated, policy:"

1. The buying team is initially managed by a distinguished agent called *firstMgr*. But any manager of this team can appoint another agent authenticated as an employee as its successor, at any time, thus losing its own managerial powers.
2. A buyer is allowed to issue purchase orders (*POs*), taking the cost of each *PO* out of its own budget – which is thus reduced accordingly – provided that the budget is large enough. The copy of each *PO* issued must be sent to the current manager.

3. An employee can be assigned a budget by the manager, and can give some of that budget to other employees, recursively. In addition, the manager can reduce the budget of any employee *e*, as it sees fit, which freezes the budget of *e*, preventing others from increasing *e*'s budget. The budget of *e*, will only be able to increase again when the manager has changed.

Messages. Item 1 of this policy is realized when the agent playing the manager role sends a message *transfer* to the employee that will be the successor. Item 2 happens when the buyer sends a *purchaseOrder(Amount)* message, where the *Amount* is the value of the purchase order that will be taken from the buyer's budget. Regarding item 3, an agent gives a budget to others by sending the message *giveBudget(Amount)*. Then, the sender's budget will be reduced by *Amount* and the addressee's budget will be increased by *Amount*. Managers can send the *removeBudget(Amount)* message. The effect of this message is to reduce by *Amount* the budget of the addressee.

XMLaw solution. XMLaw has abstractions to decompose the problem into small and more manageable pieces of information, and also to structure the steps of interactions of a complex conversational protocol. In this example, the interactions do not follow a pre-defined sequence, and the protocol is not too complex to justify decomposition into many small parts. We have specified the laws using one scene, which encapsulates the interaction protocol and a set of norms, actions and constraints. The complete specification can be found in Table 11, and the code for actions and constraints used in this specification can be found in Table 12 through Table 15. We start the explanation by describing the general syntax and dynamics of the elements. We have also provided a graphical notation of the protocol based on UML statechart diagrams, which is shown in Fig. 5.

There are five types of messages that can be exchanged between the agents. These messages are specified in line 02 to line 06. The format of the message is *message-id{sender,receiver,content}*. The symbol * denotes any value. It is also possible to manipulate variables; variables are stored in the context. Each scene has its own context, and there is also a general law context. They form a hierarchy of contexts.

The message specified in line 02 means that the sender will be assigned to the *budgetOwner* variable, the receiver can be any agent, and the content has the form *purchaseOrder(value)*, where the value will be assigned to the variable *amount*. The messages are used to activate the five transitions of the protocol (lines 09 to 13). However, transitions t1, t2, t3, and t4 have constraints that will be checked before they fire and will only fire if the constraints are satisfied. The constraints are specified in lines 14 and 15. Once the transitions fire, some of them activate actions, as can be seen from line 16 to 18. Finally, transition t2 also needs the norm specified in line 19 in order to fire.

Transition t1 controls the purchase orders stated in item 02 of the policy. In order to be activated, the constraint *enoughMoney* referred to in line 09 verifies if the sender identified by the variable *budgetOwner* (line 02) has enough money to issue the order (Table 14). Transition t2 controls the item 03 of the policy. It refers to the situation where an employee or a manager gives a budget to another employee. Then, t2 first verifies if the sender has enough money to transfer (Table 14), then checks to see if the employee that is about to receive the money is allowed to receive money as specified in policy 03. This verification is done through the norm *increaseBudgetProhibition* (line 19). This norm is given to an employee when the manager sends a *removeBudget* message and this message activates the transition t3. In XMLaw it can be seen in line 19, where t3 is the transition that activates the norm, and t4 is the transition that deactivates it. Therefore, if the norm is active, the transition t2 is not fired. If the agent has not such a norm, then the transition t2 will fire. Once t2 is fired, action *changeBudget* is activated (line 18). The code of this action can be found in Table 15. Transition t4 is activated when the manager sends a *transfer* message to an employee. In the protocol, the constraint *checkTransfer* (Table 12) guarantees that the sender of the message is in fact the manager. If the transition t4 fires, then the action *switchManager* (

Table 13) is executed, and the norm *increaseBudgetProhibition* is deactivated. The *switchManager* action updates the current manager, and once the *increaseBudgetProhibition* is not active, employees that have gained this norm, now are free again to receive budget through the message *giveBudget*.

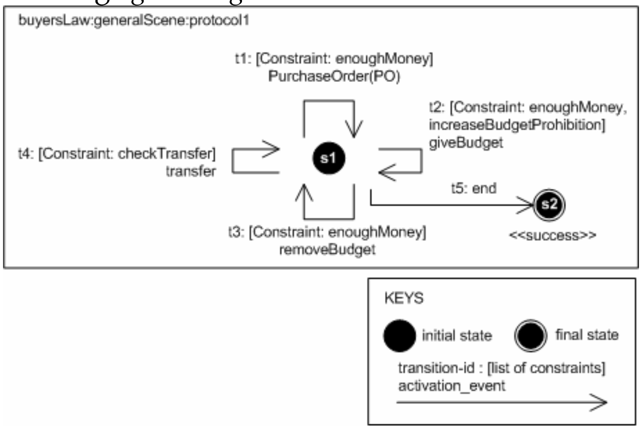

Fig. 5. Interaction Protocol

Table 11 – XMLaw Code

```
generalScene{
    PO{$budgetOwner,*,purchaseOrder($amount)}
    removeBudget{manager,$budgetOwner,removeBudget($amount)}
    giveBudget{$budgetOwner,$receiver,giveBudget($amount)}
    transfer{$manager,$employee,transfer}
    end{$sender,$receiver,end}

    s1{initial}
    s2{success}

    t1{s1->s1, PO, [enoughMoney]}
    t2{s1->s1, giveBudget, [enoughMoney],[increaseBudgetProhibition]}
    t3{s1->s1, removeBudget, [enoughMoney]}
    t4{s1->s1, transfer, [checkTransfer]}
    t5{s1->s2, end}

    enoughMoney{br.pucrio.EnoughMoney}
    checkTransfer{br.pucrio.CheckTransfer}

    forwardMessage{(t1), br.pucrio.ForwardMessage}
    switchManager{(t4), br.pucrio.SwitchManager}
    changeBudget{(t2,t3), br.pucrio.ChangeBudget}

    increaseBudgetProhibition{$budgetOwner, (t3),(t4)}
}
```

Table 12 - Constraint that verifies if the agent is in fact the current manager

```
class CheckTransfer implements IConstraint{
    public boolean constrain(ReadonlyContext ctx){
        String actualManager = ctx.get(“actualManager”);
        String currentMgr = ctx.get(“manager”);
        if (! actualManager.equals(currentMgr)){
            return true; // constrains, transition should not fire
        }
        return false;
    }
}
```

Table 13 - Action that switches the current manager to the employee

```
class SwitchManager implements IAction{
    public void execute(Context ctx){
        String employee = ctx.get(“employee”);
        ctx.put(“actualManager”,employee);
    }
}
```

Table 14 - Constraint that verifies if the one who is giving money has enough money to give

```
class EnoughMoney implements IConstraint{
    public boolean constrain(ReadonlyContext ctx){
        String budgetOwner = ctx.get(“budgetOwner”);
        double currentBudget = Double.parseString(ctx.get(budgetOwner));
        double amount = ctx.get(“amount”);
```

```
        double diff = currentBudget - amount;
        if ( diff < 0){
            // constrains, not enough money. Transition should not fire
            return true;
        }
    }
}
```

Table 15 - Action that updates budgets both for giveBudget and removeBudget

```
class ChangeBudget implements IAction{
    public void execute(Context ctx){
        String budgetOwner = ctx.get("budgetOwner");
        double currentBudget = Double.parseString(ctx.get(budgetOwner));
        double amount = ctx.get("amount");
        // update the owner`s budget (the one who is given money)
        budget.put(budgetOwner, currentBudget - amount);

        String receiver = ctx.get("receiver");
        if (receiver!=null){ // if there is a receiver
            double receiverBudget = Double.parseString(ctx.get(receiver));
            // update the receiver`s budget
            ctx.put(receiver, receiverBudget+amount);
        }
    }
}
```

Discussion. The most important part of this case study is Table 11. It is this table that contains the elements of the XMLaw conceptual model. The code in this table is mostly declarative and is concerned with high level abstractions such as interaction protocol, actions, constraints and norms. It was possible to express the rules in twenty lines of instructions, which are relatively simple to understand even for those not well versed in the XMLaw language. Even with the Java code needed to implement the actions and constraints, most of the time the designer can focus on the law specification of Table 11 and use the actions and constraints as components to achieve the desired functionality. The event-model of communication is present in most of the declarations. For example, line 19 uses event-based notification to say that the norm *increaseBudgetProhibition* is activated by the *transition_activation* event generated by the transition t3. It is deactivated by the *transition_activation* event generated by the transition t4. Another example is the transition t1 in line 09 that is activated by the *message_arrival* event generated by the message *PO*. Of course, the syntax of the language hides most of the details from the designer, and allows the event-based model to work behind the scenes. Although this case study is relatively small, it is useful to make the ideas presented in this paper more concrete. By using an existing case study it is also possible to compare this implementation with the one presented in [32].

When compared to the LGI solution in [32], the XMLaw laws in **Table 11** provide a higher-level mapping from problem specification to the solution. For example, the restriction stated in the policy item 2 "a buyer is allowed to issue purchase orders … provided that the budget is large enough" is directly mapped to the XMLaw Constraint element *enoughMoney* used in line 09.

4.2 Case Study 2: Conference Center

This case study was implemented with EI and presented in [10][33]. The example is described as follows.

"A conference takes place in a physical setting, the conference centre, where different activities take place in different locations by people that adopt different roles (speaker, session chair, organization staffer, etc.). During the conference people pursue their interests moving around the physical locations and engaging in different activities. A Personal Representative Agent (PRA) is an agent inhabiting the virtual space that is in charge of advancing some particular interest of a conference attendee by searching for information and talking to other software agents."

The example presented in [10] has structured the application in six different scenes: Information Gathering Scene, Context Scene, Appointment Proposal Scene, Appointment Coordination Scene, Advertiser Scene, and Delivery Scene. However, in [10] more details were provided for the scene Appointment Proposal, which allows us to use it as the focus on this paper.
The participants of this scene are two personal representative agents (PRA). The goal of the scene is to agree upon a set of topics for discussing during the appointment. The scene is played as follows:

1. one of the PRAs (PRA1) takes the initiative and sends an appointment proposal to the other PRA PRA2), with a set of initial topics. This proposal has a time that defines its validity (clock). We will refer to the $PRA1_x$ and to PRA2. Whenever the clock expires and $PRA1_y$ has not answered, the scene moves to s5.
2. PRA2 evaluates the proposal and can either (i) accept, (ii) decline, or (iii) send a counter proposal to $PRA1_x$ with a different set of topics. The proposal has also a time that defines its validity.
3. in turn, when PRA1 receives the counter proposal of PRA2, $PRA1_x$ evaluates this counter proposal and can also either accept, decline, or send a counter proposal to PRA2. This negotiation phase finishes when an agreement on topics is reached or one of them decides to withdraw a specific proposal.

As we have said, PRAs participate in the virtual space representing an attendee while trying to agree upon a set of topics for discussion at the appointment. Thus, when a PRA reaches an agreement for the set of topics, the PRA must inform the attendee. This is represented in XMLaw through the norm *app-notification* that can be used in other scenes to prevent agents that have not fulfilled the obligation from interacting
XMLaw solution. Fig. 6 shows a graphical representation based on UML statecharts of the interaction protocol of Appointment Proposal scene. The XMLaw specification of this scene is shown in Table 16. There are three types of messages: propose, accept and decline (lines 02 to 04). Those messages are used to fire most of the transitions (lines 08, 09, 10, 11, 13 and 14). The transition t5 is activated by the clock in line 16. This clock is activated every time transitions t1, t4 or t6 fire; and it is deactivated when there is a firing of transitions t2, t3, t4 or t5. This clock generates a *clock_tick* 5000 milliseconds after its activation. Line 15 declares the norm app-notification. This norm is given to the PRA that accepts the proposal of topics. This acceptance occurs in transition t2.

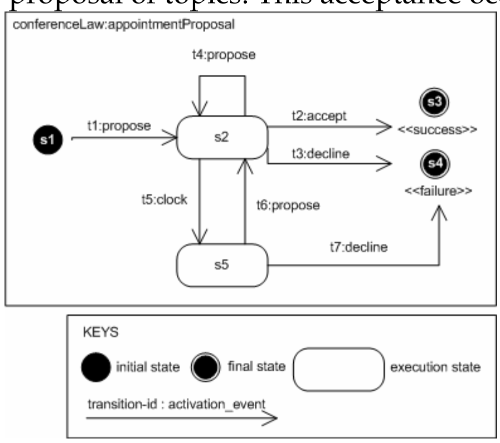

Fig. 6. Interaction Protocol

Table 16 - XMaw Code

```
01: appointmentProposal{
02:    propose{$PRA1,$PRA2,$topics}
03:    accept{$PRA1,$PRA2,$topics}
04:    decline{$PRA1,$PRA2,$reason}
```

```
    s1{initial}
    s3{success}
    s4{failure}

    t1{s1->s2, propose}
    t2{s2->s3, accept}
    t3{s2->s4, decline}
    t4{s2->s2, propose}
    t5{s2->s5, clock}
    t6{s5->s2, propose}
    t7{s5->s4, decline}

    app-notification{$PRA1, (t2),()}

    clock{5000,regular, (t1,t4,t6),(t2,t3,t4,t5)}

}
```

Discussion. When compared to the solution presented in [10], the solution presented here has some differences: (i) the set of message definitions is reused many times in the protocol, which has lead to a much simpler protocol (for example, the number of transitions was decreased from 13 to 7); (ii) because of the event-model, the clock element is plugged into the law in order to fire transitions. When compared to EI, the transition itself has a timeout element. In other words, the transition provides the functionality of the clock. This separation leads to better separation of concerns, and better reuse once clocks can be composed with other elements; (iii) as the norm is also connected to the event model, its activation is much simpler, one has only to specify which event activates the norm.

5 Discussions

In this paper we have shown that the conceptual model of XMLaw is composed of higher-level abstractions as compared to the primitives of LGI. We have also shown that the event-based notion leads XMLaw to have a more flexible model to accommodate future changes and compose the elements when compared to EI.

To be more precise, both XMLaw and LGI deal only with the exchange of messages between agents, and are not sensitive to the internal behavior of agents, and to changes in their internal state. In general, LGI is most effective for laws that are naturally local, while XMLaw is most effective for laws that are naturally global. Laws under both approaches are not intended to specify all the details of the interaction between the agents; it is merely a constraint on the interaction. From a conceptual point of view, LGI provides a state abstraction (control state), a set of events relating to communication issues, and a set of operations for manipulating the state. The state acts basically as a hashtable where terms are stored. There is no restriction over the type of terms that can be used. This lack of a restriction may lead to a great flexibility that is useful in adapting the approach to various domains. However, a small set of high level predicates could be more useful to help in the coordination and enforcement of laws without the complexity of creating new terms. The mapping of the elements from XMLaw to LGI can be seen as creating a prolog-based model of XMLaw because the events and operations provided in LGI are general and related to low level concepts. The mapping can be used if there is a need for a decentralized architecture such as LGI. It is also important to say that although global properties can be implemented in LGI, if one uses the general solution presented in [2] and referenced in Table 4, one is not making use of the decentralized nature of LGI. On the other hand, it is also possible to write laws that make use of very specific and domain dependent knowledge to synchronize states only when needed. However this approach introduces complexity for the specification of the laws and brings to the law specification concerns of distributed synchronization.

A flexible underlying event-based model as presented in XMLaw could make conceptual models of governance approaches more prepared to accommodate changes. We think that there is much space to improve the elements of the XMLaw model to make it more expressive and even easier for designing laws. One such improvement would be to incorporate the notion of norms described in [14].

To summarize, XMLaw is an alternative approach to specifying laws in open multi-agent systems that presents high level abstractions and a flexible underlying event-based model. Thus XMLaw allows for flexible composition of the elements from its conceptual model and is flexible enough to accept new elements.

We are currently extending the XMLaw model to incorporate fault tolerance techniques. The idea is to use the laws to perform error detection and then also use laws to specify the recovery strategy through error handling (rollback, rollforward, compensation) or fault handling (diagnosis, isolation, reconfiguration, reinitialization). Thus, the XMLaw model has to evolve to accommodate the concerns related to fault handling. Other work we are performing includes using the laws to collect explicit meta-data about dependability [29] using a dependability explicit computing approach. The goal is to show that our model is flexible enough to deal with very different concerns, accommodating many aspects of dependability.

ACKNOWLEDGMENTS

This work is partially supported by CNPq/Brazil under the project "ESSMA", number 5520681/2002-0 and by individual grants from CNPq/Brazil.

REFERENCES

[1] Meyer, B., The power of abstraction, reuse and simplicity: an object-oriented library for event-driven design, Festschrift in Honor of Ole-Johan Dahl, eds. Olaf Owe et al., Springer-Verlag, Lecture Notes in Computer Science 2635, 2003.

[2] Cugola, G., Di Nitto , E., and Fuggetta , A.. Exploiting an Event-based Infrastructure to Develop Complex Distributed Systems. In Proceedings of the 20th International Conference on Software Engineering (ICSE 98), Kyoto, Japan, Apr. 1998

[3] Batista , T and Rodriguez , N. Dynamic Reconfiguration of Component-Based Applications. In Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems, pages 32--39. IEEE Computer Society, June 2000

[4] Almeida, H. ; Perkusich, A.; Ferreira, G., Loureiro, E.; Costa, E. A Component Model to Support Dynamic Unanticipated Software Evolution. In: International Conference on Software Engineering and Knowledge Engineering (SEKE'06), 2006, San Francisco, USA. Proceedings of International Conference on Software Engineering and Knowledge Engineering, 2006. v. 18. p. 262-267

[5] Meier, R., Cahill, V. Taxonomy of Distributed Event-Based Programming Systems.The Computerr Journal, Vol 48 (5): 602-626, 2005

[6] Minsky, N.H., Ungureanu V.: Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems, ACMTrans. Software Engineering Methodology 9(3) 273–305, 2000

[7] Cuní, G., Esteva, M., Garcia, P., Puertas, E., Sierra, C., and Solchaga, T. MASFIT: Multi-Agent System for Fish Trading. In Proceedings of the 16th European Conference on Artificial Intelligence, 710--714, València, Spain, 2004.

[8] Ashri, R., Payne, T. R., Luck, M., Surridge, M., Sierra, C., Aguilar, J. A. R. and Noriega, P. Using Electronic Institutions to secure Grid environments. In Proceedings of Tenth International Workshop CIA on Cooperative Information Agents, 461--475, Edinburgh, Scotland, 2006

[9] Bou, E., López-Sánchez, M., Rodríguez-Aguilar, J. Norm Adaptation of Autonomic Electronic Institutions with Multiple Goals, in ITSSA journal International Transactions on Systems Science and Applications (ISSN 1751-1461 (Print); ISSN 1751-147X (CD-ROM)), vol 1, num 3, pp. 227-238, 2006

[10] Esteva, M. Electronic Institutions: from specification to development Ph. D. Thesis, Technical University of Catalonia, 2003.

[11] XMLaw Specification: version 1.0, Tech Report, PUC-Rio, 2007 – to be published

[12] Minsky, N., Law Governed Interaction (LGI): A Distributed Coordination and Control Mechanism - (An Introduction, and a Reference Manual). 2005. http://www.moses.rutgers.edu/documentation/manual.pdf - Accessed at January, 14th, 2007.

[13] Paes, R., Carvalho, G., Gatti, M., Lucena, C., Briot, J., and Choren, R. Enhancing the Environment with a Law-Governed Service for Monitoring and Enforcing Behavior in Open Multi-Agent Systems, In: Weyns, D.; Parunak, H.V.D.; Michel, F. (eds.): Environments for Multi-Agent Systems, Lecture Notes in Artificial Intelligence, vol. 4389. Berlim: Springer-Verlag, 2007, p. 221–238.

[14] Garcia-Camino, A., Noriega, P., and Rodriguez-Aguilar, J. A. 2005. Implementing norms in electronic institutions. In Proceedings of the Fourth international Joint Conference on Autonomous Agents and Multiagent Systems (The Netherlands, July 25 - 29, 2005). AAMAS '05. ACM Press, New York, NY, 667-673. DOI= http://doi.acm.org/10.1145/1082473.1082575

[15] Paes, R., Gatti, M., Carvalho, G., Rodrigues, L., Lucena, C. A Middleware for Governance in Open Multi-Agent Systems. Technical Report 33/06, PUC-Rio, 14 p. (2006)

[16] Carvalho, G., Brandão, A., Paes, R., Lucena, C.: Interaction Laws Verification Using Knowledge-based Reasoning. In: Workshop on Agent-Oriented Information Systems (AOIS-2006) at AAMAS 2006.

[17] Carvalho, G., Lucena, C., Paes, R., Briot, J.: Refinement operators to facilitate the reuse of interaction laws in open multi-agent systems. In Proceedings of the 2006 International Workshop on Software Engineering For Large-Scale Multi-Agent Systems (2006)

[18] Carvalho G., Almeida H., Gatti, M., Vinicius, G., Paes, R., Perkusich, A., Lucena, C.: Dynamic Law Evolution in Governance Mechanisms for Open Multi-Agent Systems. Second Workshop on Software Engineering for Agent-oriented Systems (2006)

[19] Arcos, J., Esteva, M., Noriega, P., Rodríguez-Aguilar, J., and Sierra, C. Environment Engineering for Multiagent Systems. Journal of Engineering Applications of Artificial Intelligence, (18):191--204, Elsevier Ltd. January, 2005.`

[20] Paes, R.B., Carvalho G.R., Lucena, C.J.P., Alencar, P.S.C., Almeida H.O., Silva, V.T.: Specifying Laws in Open Multi-Agent Systems. In: Agents, Norms and Institutions for Regulated Multi-agent Systems (ANIREM), AAMAS2005. (2005)

[21] Paes, R.B., Gatti, M.A.C., Carvalho, G.R., Rodrigues, L.F.C., Lucena, C.J.P.: A Middleware for Governance in Open Multi-Agent Systems. Technical Report 33/06, PUC-Rio, 14 p. (2006)

[22] Lindermann, G., Ossowski, S., Padget, J., Vázquez Salceda, J.: International Workshop on Agents, Norms and Institutions for Regulated Multiagent Systems (ANIREM 2005), http://platon.escet.urjc.es/ANIREM2005/ accessed in December, 2006.

[23] Domain Language Inc. The Challenge of Complexity. http://domaindrivendesign.org/, accessed in January, 2007.

[24] Murata, T. and Minsky N,. On monitoring and steering in large scale multi-agent systems. In In the Proceedings of the 2nd. International Workshop on Large Scale Multi Agent Systems, Portland Oregon, May 2003.

[25] N. H. Minsky. On conditions for self-healing in distributed software systems. In In the Proceedings of the International Autonomic ComputingWorkshop Seattle Washington, June 2003.

[26] Minsky, N. On a principle underlying self-healing in heterogeneous software. Journal of Integrated Computer-Aided Engineering, 2005.

[27] Minsky, N. H. and Rozenshtein, D. 1987. A law-based approach to object-oriented programming. In Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications (Orlando, Florida, United States, October 04 - 08, 1987). N. Meyrowitz, Ed. OOPSLA '87. ACM Press, New York, NY, 482-493. DOI= http://doi.acm.org/10.1145/38765.38851

[28] Noriega, P. "Agent mediated auctions: The Fishmarket Metaphor" Ph.D. Thesis. Universitat Autònoma de Barcelona, 1997

[29] Kaâniche, M., Laprie, J-C, Blanquart, J-P. A Dependability-Explicit Model for the Development of Computing Systems., Lecture Notes in Computer Science, Volume 1943/2000, 2004

[30] de C. Gatti, M. A., de Lucena, C. J., and Briot, J. 2006. On fault tolerance in law-governed multi-agent systems. In Proceedings of the 2006 international Workshop on Software Engineering For Large-Scale Multi-Agent Systems (Shanghai, China, May 22 - 23, 2006). SELMAS '06. ACM Press, New York, NY, 21-28. DOI= http://doi.acm.org/10.1145/1138063.1138068

[31] Rodrigues, L.; Carvalho, G.; Paes, R.; Lucena, C. Towards an Integration Test Architecture for Open MAS. In: Software Engineering for Agent-oriented Systems (SEAS 05). Uberlândia, Brasil, 2005

[32] Minsky, N., Murata, T., On Manageability and Robustness of Open Multi-agent Systems, In Software Engineering for Multi-Agent Systems II, pp. 189—206, Lecture Notes in Computer Science, 2004.

[33] Rodriguez-Aguilar, J. On the design and construction of agent-mediated electronic institutions, PhD Thesis, volume 14 of Monografies de l'Institut d'Investigació en Intel.ligència Artificial. Consejo Superior de Investigaciones Científicas, 2003

[34] Dignum, V., Vazquez-Salceda, J., and Dignum, F. 2004. A Model of Almost Everything: Norms, Structure and Ontologies in Agent Organizations. In Proceedings of the Third international Joint Conference on Autonomous Agents and Multiagent Systems - Volume 3 (New York, New York, July 19 - 23, 2004). International Conference on Autonomous Agents. IEEE Computer Society, Washington, DC, 1498-1499. DOI= http://dx.doi.org/10.1109/AAMAS.2004.20