On Fault Tolerance in Law-Governed Multi-Agent Systems

Maíra A. de C. Gatti [1], Gustavo R. de Carvalho [1], Rodrigo B. de Paes [1], Carlos J. P. de Lucena [1], and Jean-Pierre Briot[2]

[1] Software Engineering Laboratory, PUC-Rio, Rio de Janeiro, Brazil
{mgatti, guga, rbp, lucena}@les.inf.puc-rio.br

[2] Laboratoire d'informatique de Paris 6 (LIP6), Universit´e Pierre et Marie Curie, Paris, France
Jean-Pierre.Briot@lip6.fr

Abstract. The dependability of open multi-agent systems is a particular concern, notably because of their main characteristics as decentralization and no single point of control. This paper describes an approach to increase the availability of such systems through a technique of fault tolerance known as agent replication, and to increase their reliability through a mechanism of agent interaction regulation called law enforcement mechanism. Therefore, we combine two frameworks: one for law enforcement, named XMLaw, and another for agent adaptive replication, named DimaX, in which the decision of replicating an agent is based on a dynamic estimation of its criticality. Moreover, we will describe how we can reuse some of the information expressed by laws in order to help at the estimation of agent criticality, thus providing a better integration of the two frameworks. At the end of the paper, we recommend a means to specify criticality monitoring variation through a structured argumentation approach that documents the rationale around the decisions of the law elements derivation.

Keywords: Multi-Agent Systems; Open Systems; Law-Governed Approach; Dependability of Open Systems; Fault Tolerance; Requirements; Criticality; Availability, Reliability.

1 Introduction

There are many definitions in the literature for agents and, consequently, multi-agent systems. And despite their differences, all of them basically characterize a multi-agent system (MAS) as a computational environment in which individual software agents interact with each other, in a cooperative manner, or in a competitive manner, and sometimes autonomously pursuing their individual goals. During this process, they access the environment's resources and services and occasionally produce results for the entities that initiated these software agents [1]. As the agents interact in a

concurrent, asynchronous and decentralized manner, this kind of system can be categorized as a complex system [2].

The absence of centralized coordination data makes it hard to determine the current state of the system and/or to predict the effects of actions. Moreover, all of the possible situations that may arise in the execution context led us to be uncertain about predicting the behavior of agents. However, in critical applications such as business environments or government agencies, the behavior of the global system must be taken into account and structural characteristics of the domain have to be incorporated [10].

A particular issue that arises from this kind of software is: how we can ensure their dependability (which is the ability of a computer system to deliver service that can justifiably be trusted [3]) considering the reliability of critical applications and availability of these agents. There are some proposals to address such problem ([3][4][5][6], for instance, for fault tolerance and [7][8] for reliability) which have been proposed in the last few years using different approaches; each one solved a restricted problem involving dependability.

In this paper we propose an approach to increase the availability of multi-agent systems through a technique of fault tolerance known as agent replication, and to increase its reliability through a mechanism of agent interaction regulation called law enforcement mechanism. Therefore, we will combine two frameworks. The first framework, named XMLaw, manages law enforcement to increase reliability and correctness. The second framework, named DimaX, manages adaptive replication of agents in order to increase fault-tolerance. In DimaX, the decision of replicating an agent is based on a dynamic estimation of its criticality given by a criticality monitoring strategy. The agent criticality defines how important the agent is to the organization and consequently to the system. The estimation of the criticality of an agent can be based on different information, as the messages it sends or receives, or the role it plays, etc. In this paper, we will describe how we can reuse some of the information expressed by laws, and supported by XMLaw, in order to further contribute to the estimation of agent criticality, thus providing a better integration of the two frameworks. The novelty of this contribution is in the proposed combination of law-based governance and replication-based fault-tolerance, rather than in specific contributions in law-based governance or in fault-tolerance.

We also propose a means to specify the criticality monitoring strategy through a structured argumentation [24] that documents the rationale around the decisions of the law elements derivation. However, it will not be detailed in this paper. Moreover, we also provide a framework that implements the criticality monitoring variation behavior specified.

The subsequent sections are organized as follows: Section 2 presents an introduction to the agent replication-based fault tolerance for multi-agent systems, and Section 3 presents the law enforcement approach for increasing the reliability of these systems. Section 4 states a scenario for the problem description. Section 5 details the proposed solution for the problem as an integrated architecture. This architecture is the integration of both approaches presented in Section 2 and 3. Section 6 presents one of the case studies implemented to validate the concepts and the architecture. And finally, Section 7 concludes this paper and presents future works.

2 Fault Tolerance in Multi-Agent Systems: Agent Replication

The multi-agent systems deployed in an open environment, where agents from various organizations interact in the same MAS, are distributed over many hosts and communicate over public networks, hence more attention must be paid to fault tolerance.

Several approaches ([4][14][15]) address the multi-faced problem of fault tolerance in multi-agent systems. Some of them handle the problems of communication, interaction and coordination of agents with the other agents of the system. Others address the difficulties of making reliable mobile agents, which are more exposed to security problems. Some of them are based on replication mechanisms [9], and as mentioned before they have solved many problems of ubiquitous systems.

Agent replication is the act of creating one or more replicas of one or more agents, and the number of each agent replica is the replication degree; everything depends on how critical the agent is while executing its tasks. Among the significant advantages over other fault-tolerance solutions, first and foremost, agent replication provides the groundwork for the shortest recovery delays. Also, generally it is less intrusive with respect to execution time. And finally, it scales much better [9]. There is a framework, named DimaX [6], that allows dynamic replication and dynamic adaptation of the replication policy (e.g., passive to active, changing the number of replicas). It was designed to easily integrate various agent architectures, and the mechanisms that ensure dependability are kept as transparent as possible to the application. Basically, DimaX is the integration between a multi-agent system called Dima and the dynamic replication architecture for agents called DarX.

There are two cases that might be distinguished: 1) the agent's criticality is static and 2) the agent's criticality is dynamic. In the first case, multi-agent systems have often static organization structures, static behaviors of agents, and a small number of agents. Critical agents, therefore, can be identified by the designer and can be replicated by the programmer before run time.

In the second case, the agent criticality cannot be determined before run time due to the fact that the multi-agent systems may have dynamic organization structures, dynamic behaviors of agents and a large number of agents. Then it is important to determine these structures dynamically in order to evaluate agent criticality. The approach detailed in [16] proposes a way of determining it through role analysis. It could be done by some prior input from the designer of the application who specifies the roles' weights, or there would be an observation module for each server that collects the data through the agent execution and their interactions. In the second approach, global information is built and then used to obtain roles and degree of activity to compute the agent criticality.

Another way of dynamically determining these structures to evaluate agent criticality is to represent the emergent organizational structure of a multi-agent system by a graph [6]. The hypothesis is that the criticality of an agent relies on the interdependences of other agents on this agent. First, the interdependence graph is initialized by the designer, and then it is dynamically adapted by the system itself. Some algorithms to dynamically adapt and describe it are proposed in [6].

We will present here an enhancement of these approaches and it will be further described in Section 5. Basically, we improved the agent criticality calculation

through dynamic elements present during interactions with other agents. These elements will be described in the next section while the law enforcement approaches, especially the one that was chosen, are exposed.

3 Law-Governed Interaction

In open multi-agent systems the development takes place without a centralized control, thus it is necessary to ensure the reliability of these systems in a way that all the interactions between agents will occur according to the specification and that these agents will obey the specified scenario. For this, these applications must be built upon a law-governed architecture.

In this kind of architecture, enforcement that is responsible for the interception of messages and the interpreting of previously described laws is implemented. The core of a law-governed approach is the mechanism used by the mediator to monitor the conversations between agents.

Note that law-governed approaches have some relations with general coordination mechanisms (e.g., tuple-space mechanisms like Tucson [26]) in that they specify and control interactions between agents. However, the specificity of law-governed mechanisms is about controlling interactions and actions from a social (social norms) perspective, whereas general coordination languages and mechanisms focus on means for expressing synchronization and coordination of activities and exchange of information, at a lower (not social) computational level.

Among the models and frameworks that were developed to support law-governed mechanism (for instance, [7][8][17][18]), XMLaw [7] was chosen for three main reasons. First, because it implements a law enforcement approach as an object-oriented framework, which brings the benefits of reuse and flexibility. Second, it allows normative behavior that is more expressive than the others through the connection between norms and clocks. And finally, it permits the execution of Java code through the concept of actions. Thus, in this section, we explain the XMLaw description language [7] and the M-Law framework [19].

M-Law works by intercepting messages exchanged between agents, verifying the compliance of the messages with the laws and subsequently redirecting the message to the real addressee, if the laws allow it (Figure 1). If the message is not compliant, then the mediator blocks the message and applies the consequences specified in the law.

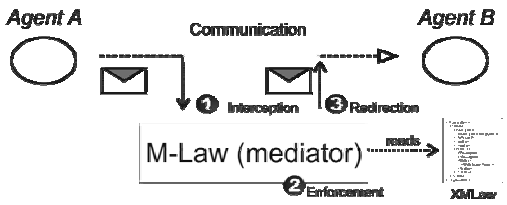

Fig. 1. M-Law Architecture.

This infrastructure, whenever necessary, can be extended to fulfill open system requirements or interoperability concerns. M-Law architecture is based on a pool of mediators that intercept messages and interpret the previously described laws.

M-Law was built to support law specification using XMLaw. XMLaw is the description language used to configure the M-Law mediator by representing the interaction rules of an open system. These rules are interpreted by M-Law that analyzes the compliance of software agents with interaction laws at runtime. Basically, interactions should be analyzed and subsequently described using the concepts proposed in the model during the design phase. After that, the concepts have to be mapped to a declarative language based on XML. It is also important to point out that agent developers from different open MASs must agree upon interaction procedure. In fact, each open MAS should have a clear documentation about the interactions' rules. By doing that, there is no need of agent developers' interaction.

Interaction's definitions are interpreted by a software framework that monitors component interaction and enforces the behavior specified by the language. Once interaction is specified and enforced, despite the autonomy of the agents, the system's global behavior is better controlled and predicted. Interaction specification of a system is also called the laws of a system. This is because besides the idea of specification itself, interactions are monitored and enforced. Then, they act as laws in the sense that they describe what can be done (permissions), what cannot be done (prohibitions) and what must be done (obligations).

Among the model elements, the outer concept is the LawOrganization. This element represents the interaction laws (or normative dimension) of a multi-agent organization. A LawOrganization is composed of scenes, clocks, norms and actions. Scenes are interaction contexts that can happen in an organization. They allow modularizing interaction breaking the interaction of the whole system into smaller parts. Clocks introduce global times, which are shared by all scenes. Figure 2 summarizes the XMLaw conceptual model, its concepts and their relations.

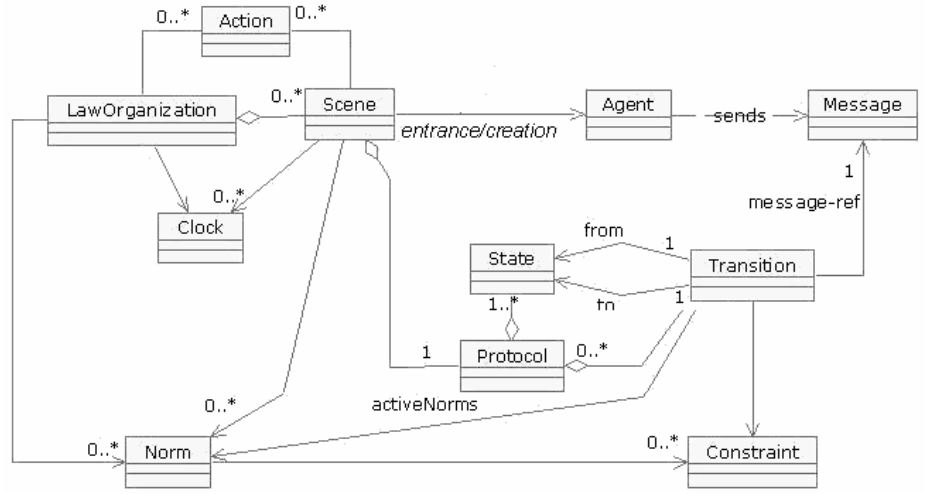

Fig. 2. Partial View of XMLaw Conceptual Model

Norms capture notions of permissions, obligations and prohibitions regarding agents' interaction behavior (as mentioned before). Actions can be viewed as a consequence of any interaction condition; for example, if an agent acquires an obligation, then action "A" should be executed.

Scenes define an interaction protocol (from a global point of view), a set of norms and clocks that are only valid in the context of the scene. Furthermore, scenes also identify which agents are allowed to start or participate in the scene.

Events are the basis of the communication among law elements; that is, law elements dynamically relate with other elements through event notifications. Basically, we can understand the dynamic of the elements as a chain of causes and consequences, where an event can activate a law element; this law element could generate other events and so on.

Furthermore, laws may be time sensitive, e.g., although an element that is active at time t1, it might not be active at time t2 ($t1 < t2$). XMLaw provides the Clock element to take care of the timing aspect. Temporal clocks represent time restrictions or controls and they can be used to activate other law elements. Clocks indicate that a certain period has elapsed producing clock-tick events. Once activated, a clock can generate clock-tick events. Clocks are activated and deactivated by law elements. Both are referenced to other law elements.

Constraints are restrictions over norms or transitions and generally specify filters for events, constraining the allowed values for a specific attribute of an event. For instance, messages carry information that is enforced in various ways. Constraints can be used for describing the allowed values for specific attributes. Constraints are defined inside the Transition or Norm elements. Constraints are implemented using Java code. The Constraint element defines the class attribute that indicates the java class that implements the filter. This class is called when a transition or a norm is supposed to fire, and basically the constraint analyzes if the received values are valid. For instance, a constraint can verify if the date expressed in the message is valid; if it is not, the message will be blocked.

The proposal here is not to detail the framework or the language, so further details can be found in [7] and [19]. The next sections will address both DimaX and XMLaw and how their integration works.

4 Problem Description

Our approach is based on the idea that the XMLaw's elements can be analyzed in order to estimate the agent criticality. It means that a norm or constraint, for example, could increase the agent criticality according to their semantic. And the law developer could specify all the elements that can increase or decrease the agent criticality while he/she is developing the laws.

To further explain our approach, we describe a scenario where two agents exchange messages in order to achieve their goals. During the interaction, they are regulated by rules that do not allow them to send some types of messages (performatives) and some other normative elements. The idea of illustrating this scenario is to find out how and which elements (norms, clocks, etc.) of the XMLaw

could improve the agent criticality analysis that is done by DimaX. And how can it be best accomplished, considering coupling, modularity and reuse of the XMLaw specification?

4.1 A first example: the Contract Net Protocol

Before starting this scenario description, we will describe a negotiation scene based on FIPA-CONTRACT-NET protocol [21]. The goal is to map a FIPA compliant protocol into a state machine protocol and discover (through this illustration) the rationale around the XMLaw protocol specification. And then, we will present the XMLaw final protocol state machine for the scenario that will be soon detailed.

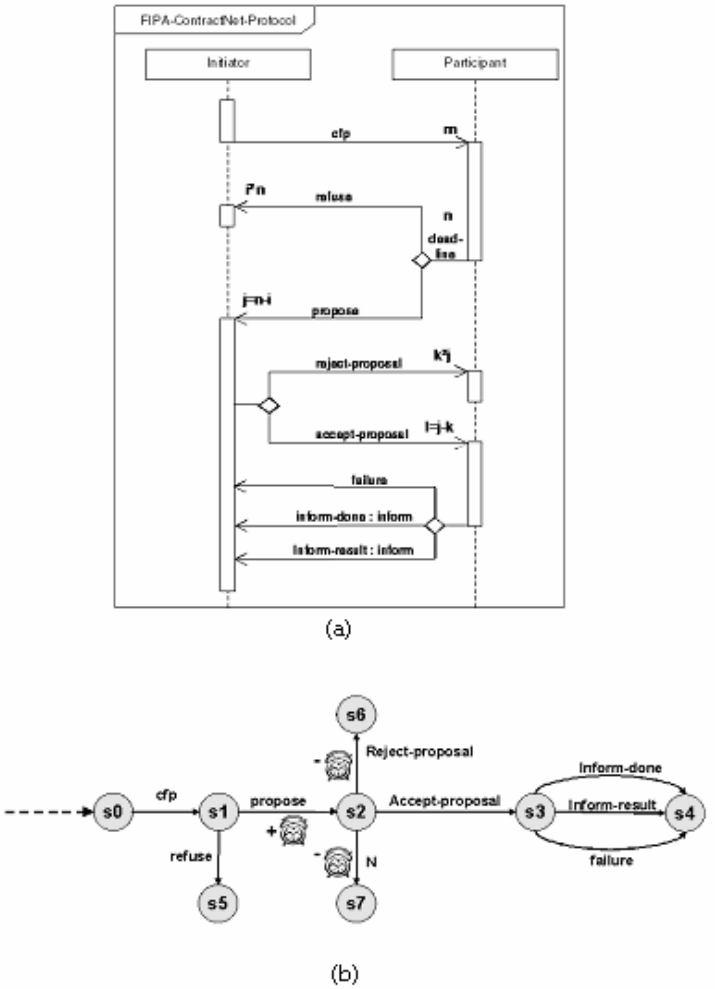

Fig. 3. a) FIPA-CONTRACT-NET protocol [21], b) FIPA Protocol State Machine

In the FIPA-CONTRACT-NET protocol, basically, the Initiator requests *m* proposals from other agents by issuing a call for proposals (*cfp*) act, which specifies the task. Participants receiving this message are viewed as potential contractors and are able to generate *n* responses. Of these, *j* are proposals to perform the task, specified as propose acts. The Participant's proposal includes the preconditions that the Participant is setting out for the task, which may be the price; the time when the task will be done, etc. Alternatively, the *i=n-j* Participants may refuse to propose. Once the deadline passes, the Initiator evaluates the received *j* proposals and selects agents to perform the task; one, several or no agents may be chosen. The *l* agents of the selected proposals will receive an accept-proposal act and the remaining *k* agents will receive a reject-proposal act. The proposals are associated with the Participant, so that once the Initiator accepts the proposal; the Participant acquires a commitment to perform the task. Once the Participant has completed the task, it sends a completion message to the Initiator in the form of an inform-done or a more explanatory version in the form of an inform-result. However, if the Participant fails to complete the task, a failure message is sent.

Now, suppose the protocol state machine shown in figure 3, where *si* represents the protocol's states during its execution and the clocks' representation are the clocks activation and deactivation for each + or -, respectively. The protocol starts with the state s0 when the Initiator solicits m proposals from other agents and it ends with the states s4, or s5, or s7, it depends on the protocol's flow.

4.2 A second example: a negotiation protocol

Considering this rationale for developing the protocol state machine, imagine a scenario where there are two agents: the customer and the seller of an institution. Suppose that an open multi-agent system exists where the agents that want to buy a product may enter or leave at any time, and that there are sellers in this institution that want to sell the product for the highest price that they can achieve. Then, we have a negotiation scene where each agent wants to succeed and there is a protocol in this scene that represents all the messages that can be exchanged and all the rules that rule this scene and the participants.

At any time, any agent can enter into the scene and initiate the protocol. If we specify this scene in XMLaw, we have to specify the protocol as a state machine, where each transition of the protocol is activated by a message sent by an agent and it can activate the other elements of XMLaw, as clocks and norms.

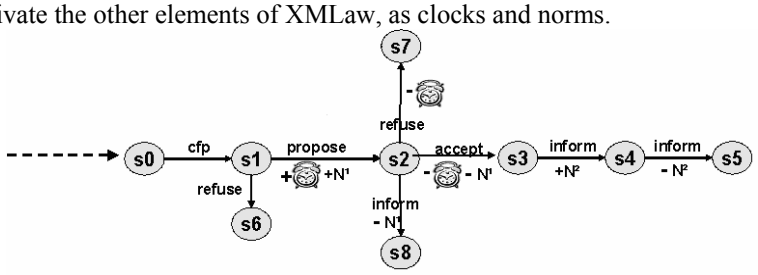

Fig. 4. Protocol State Machine Representation of the Scene Negotiation

Basically, the negotiation proceeds as follows: a customer initiates a negotiation by sending a proposal for a book to a seller. It informs the maximum price that he will pay for the book. The seller can accept the proposal or can refuse it. If he accepts, it can send proposals with lesser or equal price informed by the customer. When the customer receives the proposal, it has 2 minutes to decide if he will accept it or not. After 2 minutes, if the customer hasn't answered the seller, it can sell the product to another customer. Otherwise the seller is not allowed to sell it to anybody else. If the customer accepts it, the seller informs the bank where the payment must be made. Then the customer has the obligation of paying for the product and of informing the number of the voucher to the seller. The scene ends then when the customer informs that he paid it with the proof of payment (figure 4 and table 1).

If we consider that when an event (such as clock activation/deactivation, norm activation/deactivation, etc.) occurs during the scene execution, the agent criticality could increase or decrease, since the agent becomes more or less important; thus, each element should be taken into account in order to calculate the agent criticality in the best way. Moreover, other elements and events that might not be handled by XMLaw should be analyzed in order to evaluate how they could influence the agent criticality analysis. For instance, when an agent starts playing a role its criticality may increase or decrease.

Table 1. Protocol State Machine Description.

State		Message	Description	Event
Initial	Final			
s0	s1	Cfp	The customer starts a negotiation sending a proposal to a seller.	
s1	s1	propose	The seller accepts the customer's proposal. He sends proposals with lesser or equal price informed by the customer.	Clock activation Norm activation
s1	s6	Refuse	The seller refuses the customer's proposal and the protocol ends.	
s2	s3	Accept	The customer accepts the seller's proposal before 2 minutes.	Clock deactivation
s2	s7	Refuse	The customer refuses the seller's proposal before 2 minutes.	Clock deactivation
s2	s8	inform	The customer doesn't answer the seller and the seller informs him that he can offer the book to another customer.	Norm deactivation
s3	s4	Inform	The seller informs the customer the bank where he has to pay for the book and he has the obligation to pay in order to receive the book.	Norm activation
s4	s5	Inform	The customer informs the voucher and has the permission to receive the book.	Norm deactivation

In the context of the negotiation scene, when the customer must answer the seller if he will accept his proposal or refuse it since the clock activation event will be fired, his criticality should increase, since the seller cannot sell the product while the customer doesn't answer him. Thus, the customer is very important to the seller at this time and should not crash. Then, when the clock deactivation is fired, the customer criticality should decrease. Another situation would be the payment for the product. Since the

customer has the obligation of paying for the product when he accepts the price, his criticality should also increase. Those variations are shown in figure 5.

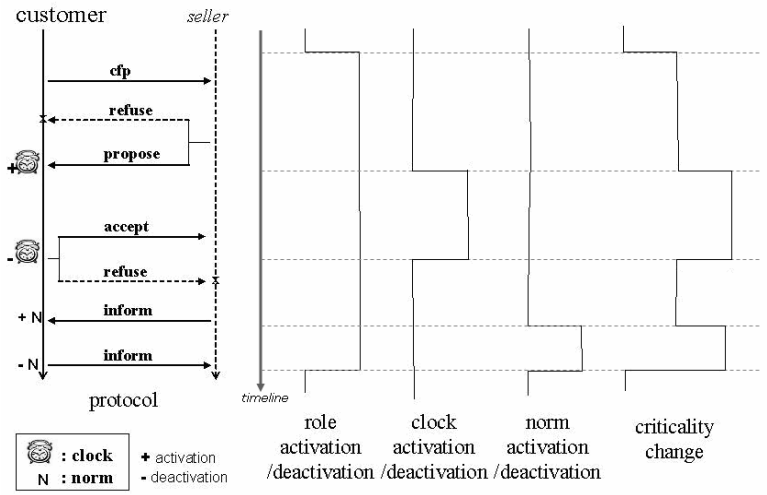

Fig. 5. Criticality variation for customer role

We can see the protocol execution on the left side of the picture. Next to it is a draft of the main criticality variation. This main result is based on the criticality variation that occurs as a result of each event, as previously mentioned. The clock's picture represents the clock activation/deactivation event and the letter N represents the norm activation/deactivation event during the protocol execution, according to the plus or minus sign that comes before the picture or letter. For instance, in an analogous manner, if we analyze the seller criticality during the scene execution, his criticality should increase when the customer proposes a price for the product because he has the obligation to answer him.

That said, it is important to highlight that the goal of this work is to combine the law-based governance with a replication-based fault-tolerance technique. This combination will improve the agent criticality estimation. This, by its turn, improves the agent replication technique of open multi-agent systems. Our proposal is that the agent criticality estimation will be done also through the events generated by the law elements. Those events may be fired during a protocol execution and can increase or decrease the agent criticality according to the type of the event and to its semantic. It could be a norm/clock/role/transition activation/deactivation event or even a message arrival event.

In the next section, we will explain how we extended both XMLaw and DimaX to attempt both the design strategies of estimating the agent criticality and its execution at runtime. We also will describe the integrated architecture developed.

5 Proposed Solution: The Integration

In this section we will present the integrated architecture and we will describe the proposed solution, first from the XMLaw and M-Law point of view, second from the DimaX point of view. At the end, we conclude describing how to use the mechanism and instantiate the resultant framework.

5.1 The Proposed Architecture

A sample scenario was created in order to illustrate the integrated architecture of both M-Law and DimaX framework (figure 6). Considering two agents: Agent A and Agent B, each one has its monitor agent called, Agent A's Monitor and Agent B' Monitor, respectively. Suppose that both are running in the same machine (host) and that each monitor will register itself in a communication port through a socket communication channel when it starts its execution.

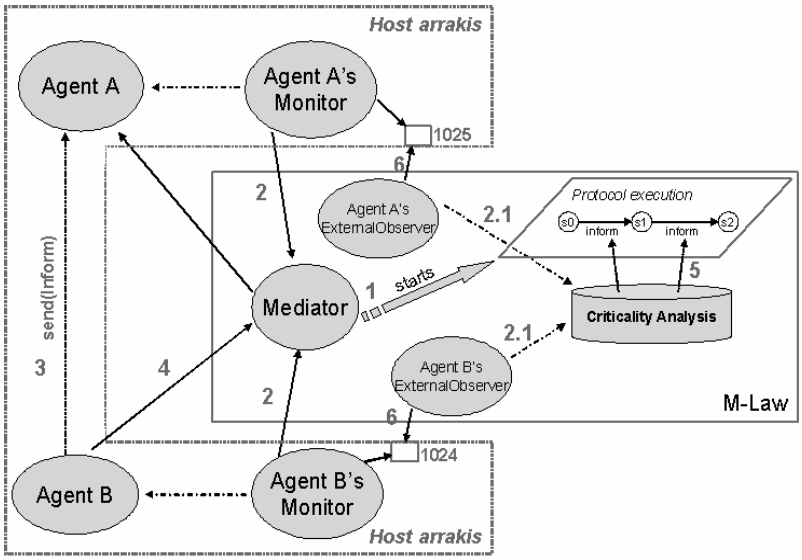

Fig. 6. The Integrated Architecture Mechanism Overview

The following flow will be executed when the agents are created and, for instance, when an interaction scene between both is started:

0. DimaX Server is started;
1. M-Law Server is started and the XMLaw file is loaded
2. DimaX monitors the agent interactions
3. The Agent B sends a message to the Agent A
4. M-Law mediator applies the enforcement

5. The criticality analysis module monitors the events and recalculate the agent's criticality. This module fires an event to be sensed by the ExternalObserver of each agent

6. The ExternalObserver listens to the events and opens a socket to send the information to the (DimaX) monitor of the agent.

Therefore, during the M-Law enforcement, whenever the component of criticality of M-Law detects to recalculate the agent criticality, it fires an event of type *update_criticality* in the scene context. The *ExternalObserver* that is listening for that event and for that agent in this scene context will send a message through socket communication to the address and port number where the monitor of that agent is listening.

5.1.1 XMLaw Extensions

We have extended XMLaw through two ways: first we added more expressivity and functionality to the Role element (Figure 7). It defines the organization's and scene's roles, and is important because the agent behavior is regulated also through the role that it plays in an organization or scene. And the second extension is the new element: CriticalityAnalysis. With the adapted Role element, when an agent requests to enter in an organization, it has to inform the role it wants to play; and when a scene is executed, the agent, if accepted, will have to play its role. An organization has one or more roles to be played by agents and an agent can play different roles in different organizations. The new XMLaw conceptual model is presented in figure 7.

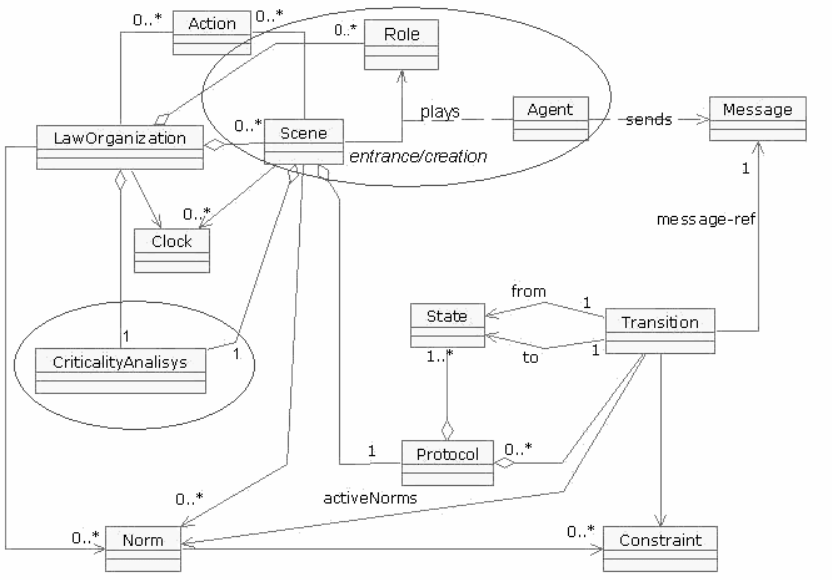

Fig. 7. XMLaw Conceptual Model

Considering the new element *CriticalityAnalysis*, it can be added in the organization or scene level. Thus, there would be a criticality analysis module for each scene in a way that, for instance, if an agent is interacting with other agents in two different scenes, and if those scenes have their criticality analysis module specified, the agent resultant criticality will be composed by the criticality variation of the scenes currently running.

The *CriticalityAnalysis* element has three elements: *Weight*, *Increases* and *Decreases*. The first one defines the weight that each event type contributes to the monitoring. Those events are the ones that increase or decrease the agents' criticality and will be referenced in the other two elements. The *Weight* element is optional because the system assumes some values to the event types that may occur. It only should be specified if the designer wants to give more or less importance to an event type than it was defined. For instance, if the law designer doesn't want to monitor the message arrival event, he should specify its value as zero.

The other two elements (*Increase* and *Decrease*) specify the necessary information for the detection and handling of the specified event by the monitoring module in order to recalculate the criticality of a given agent. The *Increases* element contains the list of events that contribute to increasing the agent criticality. And the *Decreases* element contains the list of events that contribute to decreasing the agent criticality.

Both *Increase* and *Decrease* elements are specified through three attributes and the *Assignee* element. The *event-id* attribute specifies the identification of the event to be sensed, the *event-type* attribute specifies the event type of the event defined by the *event-id* attribute, and the *value* attribute represents the associated value that the event contributes to the increasing or decreasing of the agent criticality. And, finally, the *Assignee* element contains the agent information: the agent role and a variable with the agent instance.

Table 2: Criticality specification in XMLaw

```
<CriticalityAnalysis>
  <Weight ref="role" value="0.2"/>
  <Weight ref="message" value="0"/>
  <Increases>
    <Increase event-id="customer" event-type="role_activation"
       value="0.3">
       <Assignee role-ref="customer"
          role-instance="$customer.instance"/>
    </Increase>
    <Increase event-id="seller" event-type="role_activation"
       value="0.7">
       <Assignee role-ref="seller"
          role-instance="$seller.instance"/>
    </Increase>
    <Increase event-id="time-to-decide"
       event-type="clock_activation" value="0.5">
       <Assignee role-ref="customer"
          role-instance="$customer.instance"/>
    </Increase>
    <Increase event-id="customer-payment-voucher"
       event-type="norm_activation" value="0.8">
       <Assignee role-ref="customer"
          role-instance="$customer.instance"/>
    </Increase>
    </Increases>
    …
  </Decreases>
</CriticalityAnalysis>
```

Considering the sample scenario presented in the problem description section, table 2 shows the resultant specification to the criticality monitoring of the specified scene as an example of XMLaw specification using the described elements. For instance, notice that the message arrival events will not be monitored. On the other hand, the role activation/deactivation event will be monitored with a different value (0.2).

Basically, the specification shows that, when an agent starts playing the customer role, its criticality has to be recalculated and updated by a weight of 0.3. The same happens when an agent starts playing the seller role, its criticality has to be updated by a weight of 0.7. Those actions are executed when the role activation event is fired.

5.1.2 M-Law Framework Extensions

This section presents the extensions developed in M-Law in order to implement the behavior of the criticality's monitoring module added to the XMLaw language. And it also presents the mechanism that implements this behavior.

As the M-Law is an event based framework where the protocol state moves forward through event notification, the new element *CriticalityAnalysis* had to be implemented like the other elements. Doing it by this way, it would generate and sense the system events and would minimize any overhead to the system.

That said, the *CriticalityAnalysisExecution* class implements the *IObserver* interface (figure 8). The *IObserver* interface defines the behavior of the events consumers. The consumers subscribe their interests in some event type using the *attachObserver* method. Thus, when the *CriticalityAnalysisExecution* class is instantiated, it subscribes itself for each specified event in the *Increases* and *Decreases* lists of the *CriticalityAnalysis* element, and for the event of message arrival if specified.

When the instance of the *CriticalityAnalysisExecution* object is executing and receives an event, it checks if the event data match with the specified data concerning the agent that will have its criticality updated. To this end, the *RoleReference* received is compared to the expected assignee *RoleReference*.

Once the event that occurred is the expected one, the *updateAgentCriticality* method is executed. This method receives the agent identification, the event weight value, the event value, the operation and the additional information about the event to be written by the agent monitor in the log.

This method doesn't update the criticality of the agent, it calculates the new value based on the collected data and sends to the agent monitor running in DimaX. Then, the monitor will apply one of its strategies for updating the agent criticality with the received value, combining or not with other strategies, and will set the finally criticality value which, by its turn, will be used to calculate the agent number of replicas.

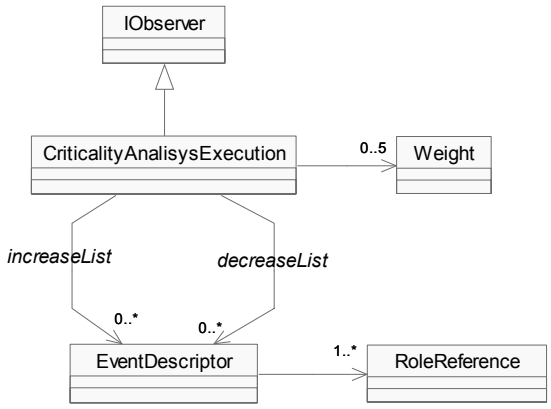

Fig. 8. Partial View of the Criticality Module Class Diagram

5.1.3 DimaX Framework Extensions

In this work we rely on the design decisions of DimaX. The model of failure considered is crash and the number of replicas depends on the criticality. However, DimaX also assumes that resources for replicas may be bounded. Furthermore, the mechanism is dynamic. That is resources for replicas may be redistributed dynamically depending on the evolution of the relative criticalities of the agents.
We extended DimaX through the same extension reasoning used to estimate through the role analysis [16][23] approach for updating the agents' criticality.

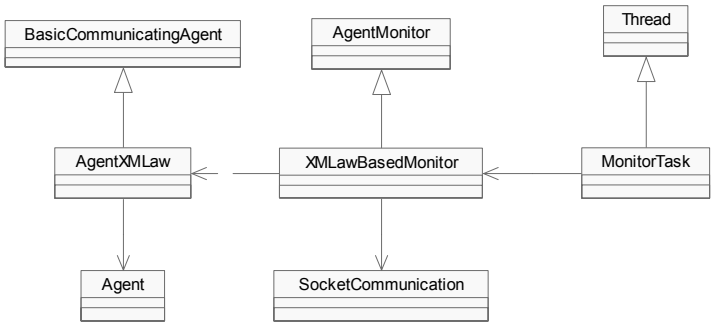

Fig. 9. DimaX Extensions Class Diagram

Figure 9 shows the class diagram of the extensions. The *BasicCommunicatingAgent* and *AgentMonitor* classes there were already in the DimaX framework. And the *Agent* class there was already in the M-Law framework. Finally, the *Thread* class is a *Java* class. We created the *XMLawBasedMonitor* class which extends the *AgentMonitor* class and implements the agent monitor behavior. It has a reference to the *AgentXMLaw* class, which is the class that implements the agent

behavior and delegates the law-enforcement tasks to the *Agent* class in a transparent way. Moreover, the *MonitorTask* is a thread that listens for the messages sent through the socket communication channel and, when a message arrives, starts the mechanism of updating the agent criticality implemented by the agent monitor through the *computeCriticalityFromXmlaw* call method.

The message received contains the agent identification, the criticality value calculated by XMLaw, the operation to be performed and the additional information about the event generated. The criticality value is the event type weight value (a_e) multiplied by the event weight value (v_e). Let the $w_i(t)$ the agent *i*'s criticality in the instant t, the final criticality value varies according to the following criteria:

– If it was generated by a Increase element:

$$\boldsymbol{wi\ (t)} = w_i\ (t) + a_e * v_e\ .$$

– If it was generated by a Decrease element:

$$\boldsymbol{wi\ (t)} = w_i\ (t) - a_e * v_e\ .$$

The result of those expressions would be combined with others results derived from criticality estimation and the degree of activity of the agent would be considered in this estimation. Finally, the calculation of the number of replicas nbi of Agent i, which is used to update the number of replicas of the domain agent, is determined as the same as before:

$$\boldsymbol{nbi(t)} = \text{rounded}(rm + w_i(t) * Rm/W)\ .$$

Where w_i is the agent criticality, W is the sum of the domain agents' criticality, rm is the minimum number of replicas which is introduced by the designer, Rm is the available resources that define the maximum number of possible simultaneous replicas.

6 Case Study

We have chosen the SELIC application to validate our approach and architecture. This system was chosen because of its unique characteristic of being an open governed distributed system regulated by a set of rules. Thus, it can be easily and directed mapped to an open law-governed multi-agent system.

The SELIC works as a mediator of the security's negotiation interactions. Concerning the negotiations, the system takes the purchase or sale commands in full or part, definitive or committed, by the necessaries proceedings to the financial movement and of custody related to the settlement of those operations, which are done one by one in real time.

We choose to implement a committed operation. There are several requirements that rule the interaction on behalf of all institutions in a committed operation, as the several types of messages that could be sent and the several behavioral that should be implemented according to the messages specified, including norms and constraints.

We choose a scenario that encloses all the law elements necessaries to the concepts proven. Figure 10 shows this scenarios and below there is an example of interaction.

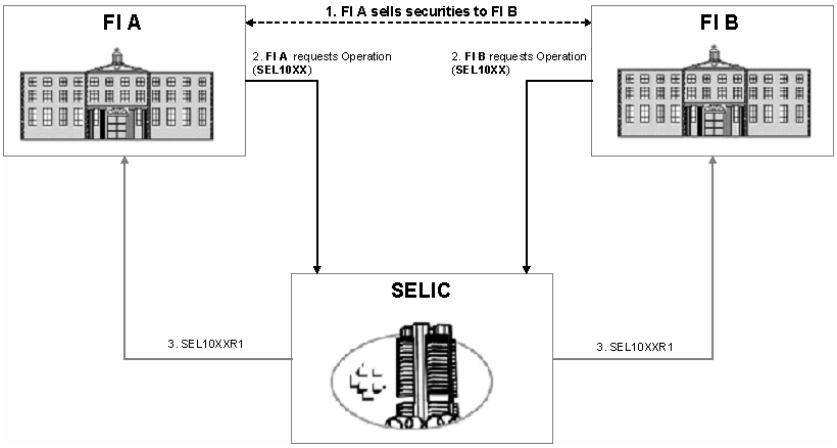

Fig. 10. SELIC Example

The financial institution A (FI A) needs to sell securities to the financial institution B (FI B) and takes the commitment of repurchasing them in the following day. It works like if FI A was taken a loan from FI B for a day.

- The SELIC notifies the financial institutions that the operations are open for negotiations (inform);
- The FI A requests the securities' sale to SELIC (request);
- The FI B request the securities' purchase to SELIC (request);
- The SELIC updates the deposit account of both institutions and informs the operation status (inform);
- In the day after, the FI A requests the securities' purchase to SELIC (request);
- The FI B requests the securities' sale to SELIC (request);
- Once again, the SELIC updates the deposit account of both institutions and informs the operation status (inform).

While those steps are executed, some constraints are also executed. As it is a committed operation, when the securities are sold, the seller acquires the obligation of repurchasing the securities in the following day. A fine will be applied to the seller every day while it doesn't repurchase the securities. After 10 days without repurchasing the securities, the financial institution is prohibited of repurchasing them again. And, concerning the buyer, it is obligated to resale the securities. While it doesn't resale the securities, the buyer will be fined daily. After 10 days, it will be prohibit interacting in the system.

Considering this scenario, we identified the events that would increase or decrease the agents' criticality. Then, the specification of the criticality monitoring was generated using the XMLaw language. For instance, we noticed that the main system threat is the possibility of the SELIC agent gets so overloaded that it could stop, fail or crash. To mitigate this risk, we analyzed the SELIC agent criticality and we implemented it through the mechanism proposed in this work.

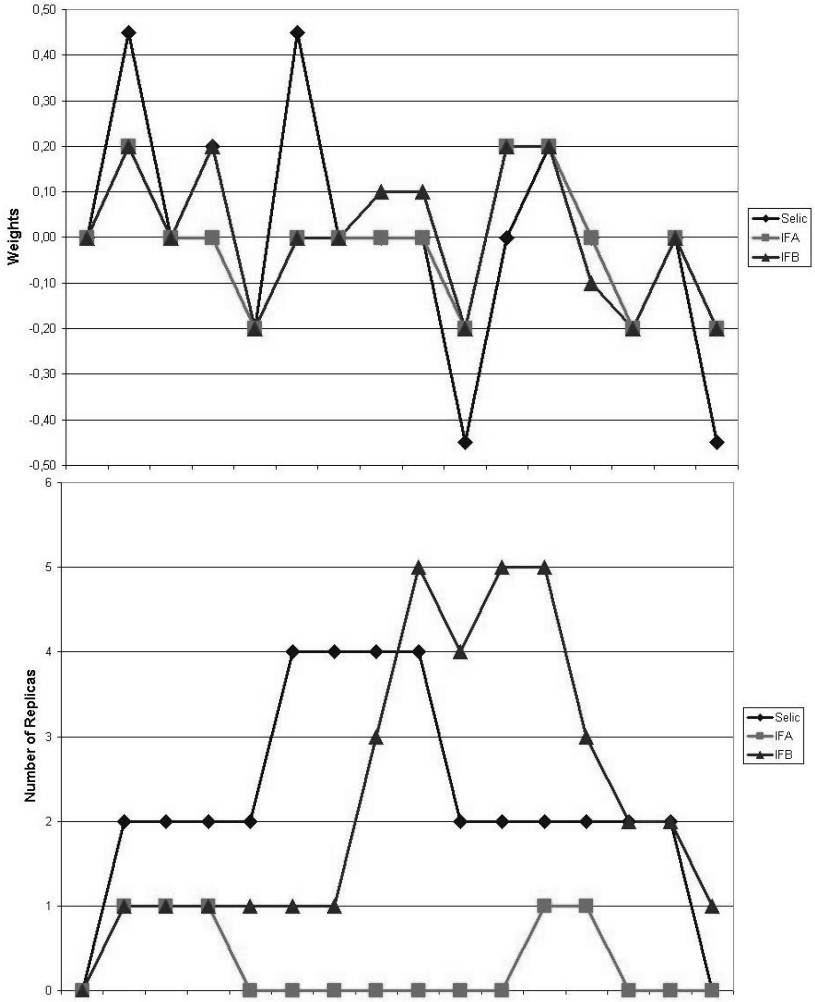

Fig. 11. SELIC Agent Criticality and Number of Replicas Variation

Figure 11 illustrates the comparing results obtained from this analyze considering the criticality variation of the three agents: the seller agent (IF A), the buyer agent (IF B), and the SELIC agent (Selic). Focusing on the SELIC agent, which can not fail otherwise no securities' negotiation would be done, its criticality monitoring created the replicas accordingly to the specification ensuring its availability when the agent became more critical to the system. The same happened to the agents playing the financial institutions roles and a particular observation point taken from the graph is that the buyer (IF B) had more replicas than SELIC because of its obligations of resale the securities. Thus, after some test-beds, the law developer would re-estimate the agents' criticalities in order to achieve the right estimation for each agent.

7 Conclusions and Future Work

This work proposed a pragmatic mechanism of estimating agents' criticality in law-governed multi-agent systems. We proposed the integration of two frameworks (M-Law and DimaX) to achieve dependability in open multi-agents system through fault tolerance and we evaluated this architecture through a real case study. During the development of this work, some initial approaches were proposed [25] and improved in order to achieve an efficient one.

We presented an extension of the XMLaw conceptual model described in Section 3 and we proposed to use new elements that help specifying the attributes concerning the agent criticality during its interaction with other agents.

We extended XMLaw with the new *CriticalityAnalysis* element. And we extended the M-Law framework to implement the monitoring of the events that should improve the criticality analysis done by DimaX. Any event considered important by the designer of the application while specifying its law can be taken into account. Finally, we extended DimaX and we integrated it with M-Law, providing another algorithm for calculating the agent's criticality.

Therefore, along these works, an important issue arose: how do we know that the criticality analyzes specification implements the real expected monitoring? Thus, we proposed the use of Law Cases [24] to help on this task and we used it on the case study presented as an evaluation of this proposal in Section 5. The Law Cases approach help to derive the law elements through a rationale that could be documented. Basically, a Law Case is a structured argument providing evidence that an open multi-agent system meets its specified dependability requirements through the rationale around the law elements derivation.

An issue to be considered is about the centralized nature of current XMLaw mediator. We are aware that it is a limitation for scalability. Hence, there is currently ongoing work to design and implemented a new distributed version.

References

1. http://agtivity.com/agdef.htm, accessed in Oct/2005.
2. Jennings, Nicholas R., An Agent-Based Approach for building Complex Software Systems, Communications of the ACM, 44(4), 35-41, April 2001.
3. Peng Xu, Ralph Deters. "Using Event-Streams for Fault-Management in MAS," IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT'04), 2004, pp. 433-436.
4. A. Fedoruk and R. Deters. Improving fault-tolerance by replicating agents. In AAMAS2002, Boulogna, Italy, 2002.
5. Fedoruk, A. and Deters, R. 2003. Using dynamic proxy agent replicate groups to improve fault-tolerance in multi-agent systems. In Proc. of the Sec. int. Joint Conf. AAMAS '03. ACM Press, New York, NY, 990-991.
6. Guessoum, Z., Faci, N., Briot, J.-P., Adaptive Replication of Large-Scale Multi-Agent Systems - Towards a Fault-Tolerant Multi-Agent Platform. Proc. of ICSE'05, 4th Int. Workshop on Soft. Eng. for Large-Scale Multi-Agent Systems, ACM Software Engineering Notes, 30(4) : 1-6, July 2005.

7. Paes, R., Carvalho, G. R., Lucena, C.J.P., Alencar, P. S. C., Almeida H.O., and Silva, V. T.. Specifying Laws in Open Multi-Agent Systems. In: Agents, Norms and Institutions for Regulated Multi-agent Systems (ANIREM), AAMAS2005, 2005.
8. Murata, T. and Minsky, N. "On Monitoring and Steering in Large-Scale Multi-Agent Systems", Proceedings of ICSE 2003, 2nd Intn'l Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS 2003).
9. Guerraoui, R. and Schiper, A. Software-based replication for fault tolerance. IEEE Computer Journal, 30(4):68--74, 1997.
10. Vázquez-Salceda, J., Dignum, V., and Dignum, F., Organizing Multiagent Systems, Autonomous Agents and Multi-Agent Systems, 11, 307-360, 2005.
11. Lussier, B. et al. 3rd IARP-IEEE/RAS-EURON Joint Workshop on Technical Challenges for Dependable Robots in Human Environments, Manchester (GB), 7-9 September 2004, 7p.
12. Laprie, J. C., Arlat, J., Blanquart, J. P., Costes, A., Crouzert, Y., Deswarte, Y., Fabre, J. C., Guillermain, H., Kaâniche, M., Kanoun, K. Mazet, C., Powel, D., Rabéjac, C. and Thévenod, P. Dependability Handbook (2nd edition) Cépaduès – Éditions, 1996. (ISBN 2-85428-341-4) (in French).
13. Avizienis, A., Laprie, J.-C., Randell, B. Dependability and its threats - A taxonomy. IFIP Congress Topical Sessions 2004: 91-120.
14. Decker, K., Sycara, K. and Williamson, M. Cloning for intelligent adaptive information agents. In ATAL'97, LNAI, pages 63–75. Springer Verlag, 1997.
15. Hagg, S. A sentinel approach to fault handling in multi-agent systems. In C. Zhang and D. Lukose, editors, Multi-Agent Systems, Methodologies and Applications, number 1286 in LNCS, pages 190–195. Springer Verlag, 1997.
16. Guessoum, Z., Briot, J.-P., Faci, N. Towards Fault-Tolerant Massively Multiagent Systems, Massively Multiagent Systems n. 3446, LNAI, Springer Lecture Note Series, Verlag, 2005, pg. 55-69.
17. Minsky, N.H., Ungureanu, V., Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems, ACM Trans. Softw.Eng.Methodol. 9 (3) (2000) 273-305.
18. Esteva, M., Electronic institutions: from specification to developement, Ph.D. thesis, Institut d'Investigació en Intelligència Artificial, Catalonia - Spain (October 2003).
19. Paes, R., Alencar, P., Lucena, C. Governing Agent Interaction in Open Multi-Agent Systems. Monografias de Ciência da Computação nº 30/05, Departamento de Informática, PUC-Rio, Brazil, 2005.
20. Weinstock, C.B., Goodenough, J.B., Hudak, J.J., Dependability Cases, Technical Note, CMU/SEI-2004-TN-016, 2004.
21. FIPA – The Foundation for Inteligent Physical Agents - Contract Net Interaction Protocol Specification http://www.fipa.org/specs/fipa00029/
22. XML Schema. http://www.w3.org/XML/Schema, last accessed in Aug, 2006.
23. Guessoum, Z., Briot, J.-P., Marin, O., Hamel, A., and Sens, P.. Dynamic and Adaptive Replication for Large-Scale Reliable Multi-Agent Systems. Software Engineering for Large-Scale Multi-Agent Systems, No 2603, p. 182–198, LNCS, Springer, 2003.
24. Gatti, M.A.C, Carvalho, G. R., Paes, R., Lucena, C.J.P, Briot, J.-P.. Structuring a Law Case for Law-Governed Open Multi-Agent Systems. Monografias em Ciência da Computação, PUC-Rio, n. MCC27/06, p. 1-34, 2006.
25. Gatti, M. A. C., Lucena, C.J.P. de, Briot, J.-P. On Fault Tolerance in Law-Governed Multi-Agent Systems. In: 5th International Workshop on Software Engineering for Large-scale Multi-Agent Systems, 2006, Shanghai. 28th International Conference on Software Engineering. New York, NY, USA : ACM Press, 2006. p. 21-27.
26. Omicini, A. and Zambonelli, F.. TuCSoN: A Coordination Model for Mobile Information Agents, *Proc. First Int'l Workshop Innovative Internet Information Systems,* June 1998.