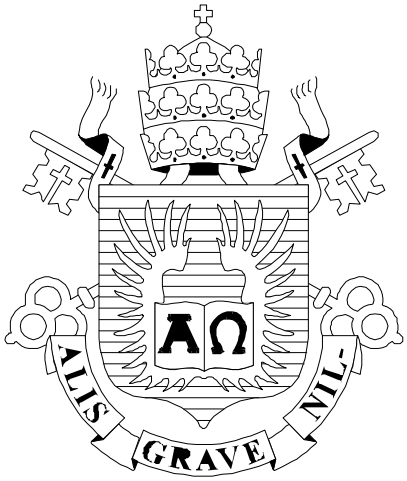

PUC

ISSN 0103-9741

Monografias em Ciência da Computação
n° 22/08

Incorporation of Dependability Concerns in the Specification of Multi-Agent Interactions by Using a Law Approach

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

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 - CEP 22451-900
RIO DE JANEIRO - BRASIL

Monografias em Ciência da Computação, No. 22/08 ISSN: 0103-9741
Editor: Prof. Carlos José Pereira de Lucena April, 2008

Incorporation of Dependability Concerns in the Specification of Multi-Agent Interactions by Using a Law Approach *

Rodrigo de Barros Paes, Carlos José Pereira de Lucena, Gustavo Robichez de Carvalho

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

Abstract. There has been a considerable amount of research using the notion of interaction laws to define the expected behavior of an open multi-agent system. In open multi-agent systems, there is little or no control over the behavior of the agents. In this paper we introduce laws as a way to support system structuring for fault tolerance. The idea is that mediators can provide very powerful means for detecting problems and allow for flexible recovery after they have been detected. The detection strategies are specified through the laws. We also discuss how some dependability attributes can be incorporated into the law specification and present the specification of two fault-tolerance techniques to illustrate our approach.

Keywords: Multiagent systems, Dependability, Interaction Laws, Governance.

Resumo. Tem havido um grande número de trabalhos de pesquisa que utilizam a noção de leis de interação para definir o comportamento esperado de um sistema multi-agente aberto. Em sistemas multi-agentes abertos existe pouco ou mesmo nenhum controle sobre o comportamento dos agentes. Neste artigo, introduz-se leis como uma técnica de tolerância a faltas. Os mediadores presentes em abordagens baseadas em leis podem ser considerados como um mecanismo poderoso para auxiliar na detecção de problemas e permitir a especificação de comportamentos flexíveis para a recuperação do sistema uma vez que um problema tenha sido detectado. As estratégias de detecção de problemas são especificadas através de leis. Neste artigo, também se discute como alguns atributos de fidedignidade podem ser incorporados a especificação de uma lei. Por fim, como exemplo, mostra-se a especificação de duas técnicas de tolerâncias a faltas utilizando a abordagem proposta.

Palavras-chave: Sistemas Multi-Agentes, Dependability, Leis de Interação, Governança.

---

* This work has been sponsored by the Ministério de Ciência e Tecnologia da Presidência da República Federativa do Brasil

In charge of publications:

Rosane Teles Lins Castilho
Assessoria de Biblioteca, Documentação e Informação
PUC-Rio Departamento de Informática
Rua Marquês de São Vicente, 225 - Gávea
22451-900 Rio de Janeiro RJ Brasil
Tel. +55 21 3527-1516 Fax: +55 21 3527-1530
E-mail: bib-di@inf.puc-rio.br

1 Introduction

There has been a considerable amount of research using the notion of interaction laws to define the expected behavior of an open multi-agent system. In open multi-agent systems, there is little or no control over the behavior of the agents. The internal implementation and architecture of agents usually are inaccessible, and different teams may have developed them but with no coordination between them. Such systems have to define behavioral rules that state what and when actions must take place. Research into interaction laws deals with this problem by explicitly specifying such rules and by providing mechanisms that check if the actual interactions conform to the specification at runtime. The mechanisms usually are implemented by either a central mediator [1][3] or by a decentralized community of mediators [2]. These mediators perform the active role of monitoring the interaction among the agents and interpreting the laws to verify if the actual system behavior is in conformance with the specifications.

In this paper we go beyond these initial ideas and introduce laws as a way to support system structuring for fault tolerance. The idea is that these mediators can provide very powerful means for detecting problems and allow for flexible recovery after they have been detected. The detection strategies are specified through the laws. Thus, we discuss how some dependability attributes can be incorporated into the law specification and present the specification of two fault-tolerance techniques to illustrate our approach.

This paper is organized as follows. In Section 2 we present a flexible Law-Governed approach called XMLaw. We use this approach throughout the examples given in this paper. Section 3 discusses how dependability concerns can be interpreted from the law specification point of view. Section 4 shows a case study where we have specified the laws to cope with dependability concerns. In Section 5 we precisely relate our research to previous work, explaining the novelty of the incorporation of dependability concerns in law specifications. Finally, in Section 6, we present some discussions about this and future work.

2 XMLaw: An Interaction Law Approach

Law-governed architectures are designed to guarantee that the specifications of open systems will be obeyed. The core of a law-governed approach is the mechanism used by the mediators to monitor the conversations between components. M-Law [3][1] is a middleware that provides a communication component, or mediator, for enforcing interaction laws. M-Law was designed to allow extensibility in order to fulfill open system requirements or interoperability concerns.

The middleware was built to support law specification using XMLaw [4][5]. XMLaw is used to represent the interaction rules of an open system specification. For readability purposes the codes written in XMLaw presented in this paper use a simplified syntax that is more compact than the one used in early XMLaw publications. These rules are interpreted by the M-Law mediator that, at runtime, analyzes the compliance of agents with interaction laws specifications. A law specification is a description of law elements which are interrelated in a way that it is possible to specify interaction protocols

using time restrictions, norms, or even time sensitive norms. XMLaw follows an event-driven approach, i.e., law elements communicate by the exchange of events.

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

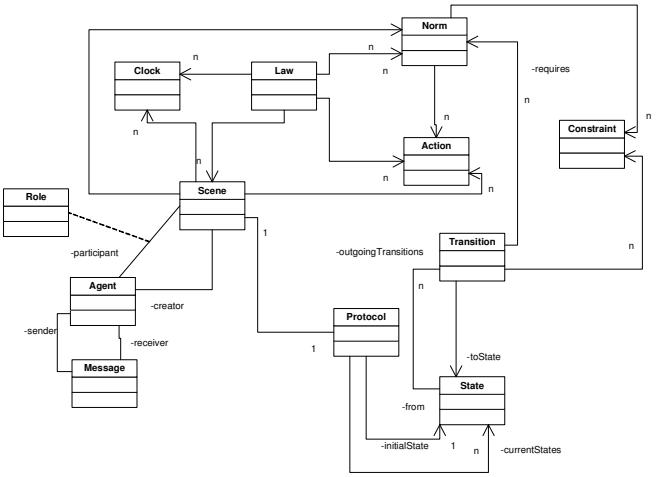

Fig. 1. XMLaw metamodel

Statically, an interaction protocol defines the set of states and transitions (activated by messages or any other kind of event) allowed for agents in an open system. Norms are jointly used with the protocol specification, constraints, actions and also temporal elements, to provide a dynamic configuration for the allowed behavior of agents in an open system. The mediator keeps information about all data regarding system execution, such as the set of activated and deactivated enforcement elements.

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

2.1 Norms

A Norm [4][5] is an element used to enable or disable agents' conversation paths. For instance, a norm can forbid an agent to interact in a negotiation scene. There are three types of norms with different semantics in XMLaw: obligations, permissions and prohibitions. The obligation norm defines a commitment that software agents acquire while interacting with other entities. For instance, the winner of an auction is obligated to pay the committed value and this commitment might contain some penalties to avoid breaking this rule. The permission norm defines the rights of a software agent at a given moment, e.g. the winner of an auction has permission to interact with a bank

provider through a payment protocol. Finally, the prohibition norm defines forbidden actions of a software agent at a given moment; for instance, if an agent does not pay its debts, it will not be allowed future participation in a scene.

The structures of the Permission (**Table 1**), Obligation and Prohibition elements are equal. Each type of norm contains activation and deactivation conditions. In **Table 1**, an assembler will receive the permission upon logging in to the scene (scene activation event called negotiation) and will lose the permission after issuing an order (event orderTransition). Furthermore, norms define the agent role that owns it through the second parameter. In Table **Table 1**, the assembler agent ($assembler) will receive the permission. Constraints and actions also can be associated with norms, but these elements will be explained later in Sections 2.2 and 2.3. Norms also generate activation and deactivation events. For instance, as a consequence of the relationship between norms and transitions, it is possible to specify which norms must be made active or deactivated for firing a transition. In this sense, a transition only could fire if the sender agent has a specific norm.

```
// norm definition
01: assemblerPermissionRFQ{permission, $assembler, (negotiation), (orderTransition)
// constraint declared in the context of the norm
02:    checkCounter{br.pucrio.CounterLimit}
// actions declared in the context of the norm
03:    permissionRenew{(nextDay), br.pucrio.ZeroCounter}
04:    rfqTransition{(rfqTransition), br.pucrio.RFQCounter}
05: } //end norm definition
```

Table 1. XMLaw specification of the permission structure

2.2 Constraints

A constraint [4][5] is a restriction over norms or transitions and, generally, it specifies filters for events, constraining the allowed values for a specific attribute of an event. For instance, messages carry information that is enforced in various ways. A message pattern enforces the message structure fields. A message pattern does not describe what are the allowed values for specific attributes, but constraints can be used for this purpose. In this way, developers are free to build constraints that are as complex as needed for their applications.

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

agent has sent until now; if it has been exceeded, the permission is no longer valid.

```
01: negotiation{
...
09:    t1{s1->s2, rfqMsg, [checkDueDate]}
...
14:    checkDueDate{br.pucrio.ValidDate}
...
20:} // end scene
```

Table 2. Constraint *checkDueDate* used by a transition

2.3 Actions

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

Since actions are also XMLaw elements, they can be activated by any event, such as a transition activation, a norm activation and even an action activation. The action structure is showed in the example of **Table 1** at lines 03 and 04 (in this example: a norm action). The class attribute of an Action specifies the Java class in charge of the functionality implementation. The first parameter references the events that activate this action and as many events as needed can be defined to trigger an action.

2.4 XMLaw for Dependability

The flexibility achieved by using the event-driven approach at a high-level of abstraction is not present in the other high level approaches [6][7]. The advantages claimed in favor of the use of events as a modeling element are also present in LGI [2], however at a low level of abstraction. A flexible underlying event-based model as presented in XMLaw can allow conceptual models for governance to be more prepared to accommodate changes. This is specially needed when we consider using the law-approach to deal with new concerns not considered in its original specification, such as dependability. For this reason, we have used XMLaw to specify and implement our case studies..

3 Laws and Dependability

Dependability of a system can be defined as the ability to avoid service failures that are more frequent and more severe than is acceptable [8]. Dependability is an integrating concept that encompasses the following attributes [8]:

- availability: readiness for correct service.
- reliability: continuity of correct service.
- safety: absence of catastrophic consequences on the user(s) and the environment.
- integrity: absence of improper system alterations.
- maintainability: ability to undergo modifications and repairs.

Many means have been developed to attain the various attributes of dependability, namely:

- Fault prevention: means to prevent the occurrence or introduction of faults.
- Fault tolerance: means to avoid service failures in the presence of faults.
- Fault removal: means to reduce the number and severity of faults.
- Fault forecasting: means to estimate the present number, the future incidence and the likely consequences of faults.

The XMLaw approach was structured in such a way that we can discuss how to incorporate these means into the law specification. The main benefit of doing this is to reuse the infrastructure of laws (the mediator and the language) to explicitly specify strategies to achieve dependability. As follows, we discuss how the means can be interpreted from the law point of view.

Fault prevention - Prevention of development faults is an aim for software development methodologies (e.g., information hiding, modularization, use of strongly-typed programming languages). Improvement of development processes in order to reduce the number of faults introduced in the produced systems is a step further in that it is based on the recording of faults in the products and the elimination of the causes of the faults via process modifications [8]. One of the problems that leads to faults is an ill-defined or ambiguous requirement specification. The law specification is in fact a precise specification of the expected behavior of the system as a whole. This specification can be used to (i) guide the development of the individual agents that compose a system; (ii) guide the development of test scripts concerning the integration among the agents; (iii) and to act as execution assertions at execution time. All of these factors can be integrated into an already existing development process. For example, activities of a development process can include: specification of use cases, specification of interaction laws, development of agents, agents test using the laws, and so on. Moreover, a well structured and extensively used law also can prevent systems from service failure. In [9][10][11], XMLaw was used to implement frameworks of laws. The idea is to prevent faults using the same law in many different application instances, in a way similar to object oriented frameworks.

Fault tolerance - Fault tolerance techniques basically are composed of two phases: error detection and error recovery. The mediators used in the law-governed approaches can provide immense support for detecting erroneous situations. We adopted the definition of error from [8], where an error is defined as the part of the total state of the system that may lead to its subsequent service failure.

Usually, the mediators are implemented as a middleware that intercepts the desired communication among the agents and acts according to the law specification. Subsequently, it is possible to write laws that are concerned with the detection of errors. For example, in XMLaw some possible sources of faults can be:

- The law specification itself that may not represent how the system is expected to behave: i.e., the developer wrote a wrong law. Consequently, this law can lead to a service failure, or failure for short.

- In XMLaw it is possible to specify external java components (actions and constraints) that will be invoked when required by the law. However, these components can contain programming faults, which can lead to failures.

- The interaction among the agents does not occur the way it has been specified in the law. In some cases, the non-conformance with the laws can mean an error situation generated by some agent fault. The laws can be used to detect and to specify strategies to deal with such situations. The XMLaw provides a set of events that can be listened to detect error situations, such as: (i) *message_not_compliant*. This event occurs when the mediator receives a message from the agent that does not match the expected message; (ii) *constraint_not_satisfied*. This event occurs when a constraint does not allow a certain interaction; (iii) agents trying to enter in scene where they do not have permission to enter; (iv) when a clock generates a *clock_tick* event it may mean that a certain agent that was supposed to send a message is not available.

Besides, the error detection situations discussed above, it also is possible to establish recovery strategies by performing error handling or fault handling. The case study presented in Section 4 shows examples of some recovery strategies implemented using the laws.

Fault removal - Some examples of fault removal techniques are inspections, model checking and testing. In [12], we have presented a test case based approach and an architecture to generate test reports by using XMLaw. Since the laws specify the expected behavior of the whole system, similar to mock objects [13], the idea is to write mock agents that implement the behavior needed by the test cases. In this manner, the developer can test the real agents while interacting with the mock agents.

Fault forecasting - Fault forecasting mainly aims at identifying, classifying and ranking the events that would lead to systems failures. In [14], XMLaw was applied to identify the criticality of agents at runtime. When an agent becomes too critical, in order to prevent the system from service unavailability, the laws specify actions that invoke a replication mechanism to create replicas of the most critical agents.

As discussed above, the laws and the mediator architecture can provide a suitable method of incorporating dependability concerns. Although we have discussed many attributes of dependability, in the next section we narrow the focus of the discussion and present a sales system example to discuss reliability issues. The idea is to show that the law specification can incorporate fault-tolerance techniques in order to support the system in the continuity of correct service.

4 Implementing Fault Tolerance Strategies

In this section, we show how designers can use the laws to incorporate dependability concerns to the specification of laws. We present a case study based on the sales control system presented in [15], and show how we implemented the requirements of this system. More specifically, we show how two forward recovery strategies shown in **Fig. 2** were specified through the laws.

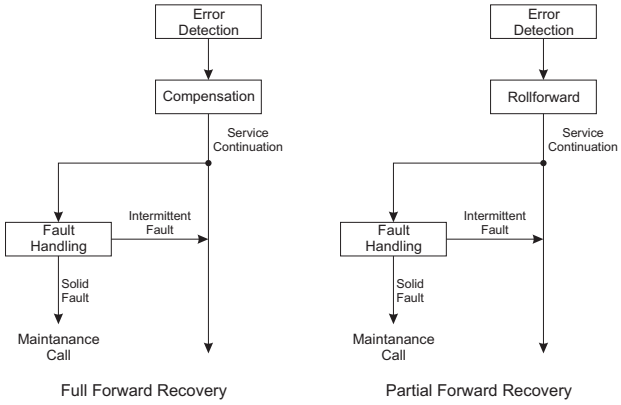

Fig. 2. Forward Recovery Strategy [8]

The sales control system consists of a database agent, a set of control points and a set of sales points, as illustrated in **Fig. 3**. Its main function is to maintain a database describing all the products to be sold so that many distributed sales points can obtain the correct prices of the items selected by the customers. Several control points provide interfaces that allow the human managers of the system to update the product information in the database at run time. We assume that such updating is regarded as a very critical activity and consequently, to guard against fraud, the policy is that two human managers, one of whom is at a senior level, have to be involved in and agree to any such updating. Thus, it will be necessary to update the data cooperatively from the control points. Such updates must also be atomic with respect to sales points that may be querying the database at the same time. Hence, an item is not really deleted or added to the database unless the corresponding action commits successfully.

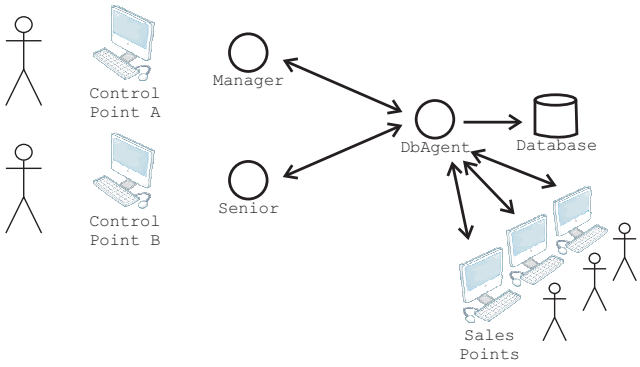

Fig. 3. Sales control system

The complete law specification is presented in **Table 3**. It will be described in detail while discussing the three scenarios below: requirement 1, situation 1 and situation 2.

REQUIREMENT 1: UPDATES MUST BE ATOMIC.

The usual approach for solving this problem would be by applying a backward recovery in the event there is some problem with the second manager confirmation. In this case study, we have approached this problem through the combined use of the interaction protocol, actions and constraints. The interaction protocol shown in **Fig. 4** defines two main paths: transitions {t1, t2} or {t3, t4}. The path {t1, t2} means the senior manager has made the first update, and the second path means the senior manager has made the second update. In both cases, when the first transition fires (t1 or t3), the action keepContent is invoked. This action stores the content of the update in the context of the scene, so this content can be used further. In fact, this content is used by the constraint checkContent. This constraint verifies if the content of the second update is equal to the previous content. If so, then the transition (t2 or t4) is finally fired and the dbAgent atomically updates the database.

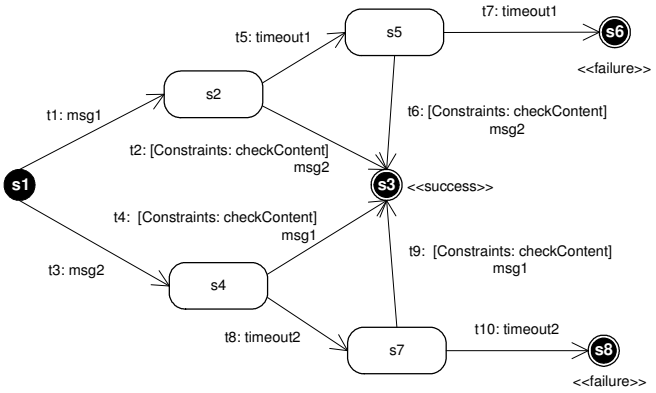

Fig. 4. Interaction Protocol

```
01:updateProductInformation{
02:   msg1{senior,dbAgent,$productInfo1}
03:   msg2{(senior | manager),dbAgent,$productInfo2}

04:   s1{initial}
05:   s3{success}
06:   s6{failure}
07:   s8{failure}

08:   t1{s1->s2, msg1}
09:   t2{s2->s3, msg2, [checkContent]}
10:   t3{s1->s4, msg2}
11:   t4{s4->s3, msg1, [checkContent]}
12:   t5{s2->s5, timeout1}
```

```
13:  t6{s5->s3, msg2, [checkContent]}
14:  t7{s5->s6, timeout1}
15:  t8{s4->s7, timeout2}
16:  t9{s7->s3, msg1, [checkContent]}
17:  t10{s7->s8, timeout2}

// Clocks
18:  timeout1{120000, periodic, (t1), (t2, t6)}
19:  timeout2{120000, periodic, (t3), (t4, t9)}

// Constraints
20:  checkContent{br.pucrio.CheckContent}

// Actions
21:  keepContent{(t1,t3), br.pucrio.KeepContent}
// Actions for fault handling
22:  handleTimeout{(t7,t10), br.pucrio.TimeoutHandler}
23:  handleDifferentContent{(checkContent), br.pucrio.DifContentHandler}
24:  warnManagerBroadcast{(t5,t8), br.pucrio.Retry}
25:}
```

Table 3. Law Specification

SITUATION 1: THE SECOND MANAGER DOES NOT ANSWER.

As the data must be updated cooperatively, the second manager is strictly necessary to commit the operation. In this case, we chose to apply a full forward recovery for when there is no update confirmation coming from the second manager. As can be seen in **Fig. 2**, there are three main activities, namely: error detection, compensation and fault handling. The law specifies how to perform these activities. **Error detection** is accomplished by perceiving that the second manager is not answering. The clocks at line 18 and 19 are activated when the message from the first manager is sent. Then, they count 2 minutes (120000 milliseconds), which is the amount of time that the second manager needs to send the second update message. If the second manager does not answer during this time, the clock generates a *clock_tick* event. By capturing this clock tick we are able to perceive when the error (in this case, the manager is not answering) has occurred. Then, after detecting the error, we can perform a **compensation** strategy. In our case, the strategy is very simple. It sends a broadcast message to all agents warning that there is an update pending due to the lack of a manager confirmation. This is done

through action *warnManagerBroadcast* at line 24. This action is activated just when transitions t5 or t8 are fired in consequence of the *clock_tick* event. The clock is declared as periodic, which means that it remains cyclically generating events every two minutes until it becomes inactive through transitions t2, t6, t4 or t9 (lines 18 and 19). Therefore, managers have two more minutes to answer the broadcast message sent by the *warnManagerBroadcast* action. If any manager answers the broadcast message with an update confirmation, then transitions t6 or t9 are fired and the protocol finishes successfully. Otherwise, if there is still no answer from the manager, there is a need for **fault handling**. This case is handled by the action *handleTimout* at line 22. This action sends a message to all agents involved in the conversation by saying that the second manager has not answered and, therefore, each agent can perform its own forward recovery strategy.

Although many complexities have been omitted for the sake of simplicity and brevity, the example is sufficiently detailed to illustrate how the laws could incorporate dependability concerns. Specifically, in this case, this is accomplished by specifying a full forward recovery strategy, through error detection, compensation and fault handling.

SITUATION 2: MANAGERS SEND DIFFERENT UPDATE CONTENTS.

In order to confirm an update message from a first manager, the second manager must send another message with exactly the same content as the first manager's message. In this case, we propose a very simple fault tolerance strategy. First, we conduct error detection through constraints and actions, and afterwards we conduct fault handling to make the agents involved in the conversation become aware of the failure. Regarding **error detection,** the action *keepContent* stores the content of the first message in the context, and the constraint *checkContent* checks if the content of the second message is equal to the first message. If the constraint *checkContent* discovers the content is not the same, then it generates the event *constraint_activation*. This event is captured by the action *handleDifferentContent*. This action basically does **fault handling,** informing all participant agents there was a wrong content. It gives the managers another opportunity to send the correct message.

In fact, this strategy performs a partial forward recovery. It detects the error, keeps the system in a safe state (note neither transitions t2 nor t4 are fired, because once contents are determined to be different, the constraint does not permit a transition to fire), performs a fault handling procedure and, in case the second manager sends another message with the same content, the protocol finishes successfully.

As previously stated in situation 1 above, also in the second situation it has been possible to show that laws may incorporate dependability concerns through their specification.

5 Related Work

Minsky [2] proposes a coordination and control mechanism called law governed interaction (LGI). This mechanism is based upon 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. However, when it is necessary to have a global view of the interactions, the decentralized enforcement demands state consistency protocols, which may not be scalable. In contrast, M-Law uses XMLaw, which provides an explicit conceptual model and focuses on different concepts, such as Scenes, Norms and Clocks. In other words, in our opinion, LGI design is aimed primarily at decentralization and XMLaw design is aimed primarily at expressivity, flexibility and at possibilities for specialization [9]. A current limitation of XMLaw is the centralization of the mediator. A work in progress is the investigation of how XMLaw specifications could be compiled into decentralized LGI mediators. In this way, LGI can be viewed as having the basic foundation to build higher-level elements, such the ones in XMLaw. Moreover, by using M-Law, it is possible to extend the framework hotspots and introduce new components, which represent concepts in the conceptual model; and change the communication mechanism.

From the point of view of dependability, LGI has a strong emphasis on security and trust. Its architecture encompasses certification authorities, cryptography and a set of operations for this means. However, to the best of our knowledge LGI has not explicitly incorporated reliability, fault handling and other issues discussed in this paper.

The Electronic Institution (EI) [6] is another approach that provides support for interaction laws. An EI has a set of high level abstractions that allows for the specification of laws using concepts such as agent roles, norms and scenes. However, EI also does not explicitly approach dependability concerns.

6 Discussions

In this paper, we have discussed the relationship between interaction laws and dependability concerns. We have presented a law-based approach called XMLaw. We showed how XMLaw could be used to implement fault-tolerance strategies.

Using laws to specify dependability concerns allows for reuse of all the infrastructure for monitoring and enforcement present in the law approaches. Besides, the dependability is defined in a precise and declarative manner.

The event-driven approach of XMLaw has contributed to specify dependability questions in a flexible way. It allows composing uncoupled elements, such as transitions, norms and clocks.

We are currently conducting some experiments using XMLaw to enable Dependability Explicit Computing (DepEx) [16]. DepEx treats dependability metadata as first-class data. We are using XMLaw to collect domain-specific metadata and subsequently use it to aid design-time and run-time decision-making.

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] R. Paes, M. Gatti, G. Carvalho, L. Rodrigues, and C. Lucena, A middleware for governance in open multi-agent systems," PUC-Rio, Tech. Rep. MCC 33/06, 2006, http://wiki.les.inf.puc-rio.br/uploads/8/87/Mlaw-mcc-agosto-06.pdf.

[2] N. H. Minsky and V. Ungureanu, Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems," *ACM Trans.Softw. Eng. Methodol.*, vol. 9, no. 3, pp. 273--305, 2000.

[3] R. Paes, G. Carvalho, M. Gatti, C. Lucena, J.-P. Briot, and R. Choren, *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, p. 221–238, 2007

[4] R. Paes, G. Carvalho, C. Lucena, P. Alencar, H. Almeida, and V. Silva, Specifying laws in open multi-agent systems," in *Agents, Norms and Institutions for Regulated Multiagent Systems - ANIREM*, Utrecht, The Netherlands, July 2005.

[5] R. Paes, G. Carvalho, and C. Lucena, Xmlaw specification: version 1.0," PUC-Rio, Rio de Janeiro, Brasil, Tech. Rep. to appear, 2007.

[6] M. Esteva, Electronic institutions: from specification to development," Ph.D.dissertation, Institut d'Investigaci en Intel.ligncia Artificial, Catalonia - Spain, October 2003.

[7] V. Dignum, J. Vzquez-Salceda, and F. Dignum, A model of almost everything: Norms, structure and ontologies in agent organizations," in *Third International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS'04)*, vol. 3, 2004.

[8] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, Basic concepts and taxonomy of dependable and secure computing," *IEEE Trans. on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11--33, Jan. 2004.

[9] G. Carvalho, C. Lucena, R. Paes, and J.-P. Briot, Refinement operators to facilitate the reuse of interaction laws in open multi-agent systems," in *SELMAS '06: Proceedings of the 2006 international workshop on Software engineering for large-scale multi-agent systems*. New York, NY, USA: ACM Press, 2006, pp. 75--82.

[10] G. Carvalho, Frameworks for open multi-agent systems," Doctoral Mentoring at AAMAS, 2006.

[11] G. Carvalho, C. Lucena, R. Paes, J.-P. Briot, and R. Choren, A governance framework implementation for supply chain management applications as open multi-agent system," in *7th International Workshop on Agent-Oriented Software Engineering (AOSE-2006)*, 2006.

[12] L. F. Rodrigues, G. Carvalho, R. Paes, and C. Lucena, Towards an integration test architecture for open mas," in *Software Engineering for Agent-oriented Systems (SEAS 05)*, Uberlndia, Brazil, 2005.

[13] T. Mackinnon, S. Freeman, and P. Craig, Endo-testing: Unit testing with mock objects," in *eXtreme Programming and Flexible Processes in Software Engineering - XP2000*, 2000.

[14] M. A. de C. Gatti, C. J. P. de Lucena, and J.-P. Briot, On fault tolerance in law-governed multi-agent systems," in *SELMAS '06: Proceedings of the 2006 international*

workshop on Software engineering for large-scale multi-agent systems. New York, NY, USA: ACM Press, 2006, pp. 21--28.

[15] J. Xu, B. Randell, A. Romanovsky, C. Rubira, R. Stroud, and Z. Wu, Fault tolerance in concurrent object-oriented software through coordinated error recovery," *ftcs*, vol. 00, p. 0499, 1995.

[16] M. Kaniche, J.-C. Laprie, and J.-P. Blanquart, A dependability-explicit model for the development of computing systems," in SAFECOMP '00: Proceedings of the 19th International Conference on Computer Safety, Reliability and Security. London, UK: Springer-Verlag, 2000, pp. 107--116.