



Rodrigo de Barros Paes

**Governança de Sistemas Multi-Agentes Abertos com
Fidedignidade**

Proposta de Tese de Doutorado

Proposta de Tese apresentada ao Programa de Pós-Graduação em Informática da PUC-Rio como requisito parcial para obtenção do grau de Doutor em Informática.

Orientador: Carlos José Pereira de Lucena

Rio de Janeiro, 20 de Junho de 2007

Sumário

1 Introdução	III
1.1. Problema	V
1.2. Limitações de Abordagens Existentes	V
1.3. Solução Proposta	VI
1.4. Contribuições	VII
1.5. Organização da Tese	VII
2 Referências	X

1

Introdução

A tecnologia de agentes vem sendo considerada como adequada para lidar com a heterogeneidade, adaptabilidade e a característica aberta de alguns sistemas distribuídos [1-3]. Em sistemas distribuídos abertos, os agentes estão imersos em um ambiente altamente imprevisível, podem ser não-cooperativos e, freqüentemente, os outros agentes que compõem o sistema não são conhecidos a priori. O comportamento global deste tipo de sistema emerge da interação entre os agentes e, portanto, não pode ser totalmente previsto. Embora este tipo de comportamento seja aceitável ou mesmo desejável, em algumas aplicações, isto pode levar a falhas no sistema. Isto é especialmente verdade em domínios tais como comércio eletrônico [4, 5], cadeia de fornecimento [6] e aplicações médicas [7].

Muitos pesquisadores vêm usando a idéia de instituições eletrônicas para alcançar níveis maiores de previsibilidade e confiabilidade nas organizações de agentes. Desta forma, a natureza aberta dos sistemas multi-agentes pode ser melhor gerenciada. Instituições eletrônicas são uma metáfora baseada nas instituições que existem na sociedade humana. Estas instituições são estabelecidas para regular as interações entre as partes que estão realizando alguma atividade. Uma vez que as interações são especificadas pelas instituições eletrônicas, é preciso que haja uma infra-estrutura de software que garanta que os agentes que fazem parte da instituição interajam de acordo com as especificações. Desta forma, é possível alcançar níveis maiores de previsibilidade e, conseqüentemente, diminuir a probabilidade de falhas no sistema. Neste contexto, vários modelos de instituições eletrônicas foram propostos [8-12].

Neste trabalho, utiliza-se o termo governança para englobar o conjunto de abordagens que objetivam estabelecer e verificar alguma forma de estrutura, conjunto de normas ou convenções de comportamento para articular ou restringir as interações dos agentes. Desta forma, as instituições eletrônicas são consideradas como uma abordagem de governança.

Por outro lado, artigos recentes enfatizam que software continua sofrendo pela falta de qualidade [13-15]. Sabe-se que cerca de 50% do software tornado disponível

contém falhas não triviais [16]. Ainda assim, é sabe-se que, independentemente do rigor com que seja desenvolvido, todo software conterá falhas. Conseqüentemente, a ocorrência de falhas de uso, de hardware ou do próprio software é um fato com o qual se precisa conviver sem que, no entanto, estas falhas levem a um nível de danos inaceitável.

Este problema na produção de software ocorre mesmo quando se considera software com arquiteturas convencionais. A tendência disto é se agravar cada vez mais já que o software é desenvolvido por equipes geograficamente distribuídas e operam de forma distribuída (várias CPUs contendo partes heterogêneas do software operando sobre uma única aplicação, por exemplo em sistemas grid). Outra razão para que o problema se torne um desafio crescente é o fato dos sistemas de software tornarem-se cada vez mais complexos (volume e abrangência da funcionalidade e requisitos de qualidade mais restritivos), precisarem estar disponíveis por mais tempo (sistemas 24h / 7 dias por semana) e serem utilizados por pessoas sem garantia de treinamento.

Em virtude da crescente participação de software na sociedade, torna-se cada vez mais necessário assegurar que ele seja fidedigno. Desenvolver software fidedigno é bem mais amplo do que incorporar tolerância a faltas e assegurar corretude, entre outras propriedades. O entendimento da noção de fidedignidade¹ depende do serviço que o software oferece. Em alguns serviços como o comando e controle, uma falha pode ter conseqüências catastróficas. Já em outros contextos, como por exemplo a busca de informação na Web, um resultado incorreto (falsos positivos ou ausência de respostas relevantes) é tolerável desde que isto ocorra com uma frequência baixa. Entretanto, não é tolerável que um sistema crítico interrompa a sua execução, a invasão de privacidade por programas maliciosos, ou ainda um aplicativo apresentar riscos à segurança de quem o usa. Ou seja, um software é dito fidedigno quando se pode depender do mesmo através de verificações formais ou informais assumindo riscos de danos compatíveis com o serviço prestado pelo software [16].

A hipótese apresentada neste trabalho é que a incorporação de mecanismos de fidedignidade em uma abordagem de governança baseada em leis de interação pode ser utilizada para a construção de sistemas multi-agentes abertos com um nível de fidedignidade adequado. Esta hipótese foi baseada na percepção de que as abordagens de governança atuais focam na especificação e verificação do comportamento esperado

¹ A palavra fidedignidade foi a tradução encontrada para o termo inglês dependability.

em um sistema. Entretanto, atributos de fidedignidade como tolerância a faltas não são levados em consideração. A incorporação de aspectos de fidedignidade em uma abordagem de governança tem como principal benefício a geração de uma tecnologia integrada que possui os vantagens de uma abordagem de governança e ao mesmo tempo lida com especificações de meios para alcançar maiores níveis de fidedignidade.

Desta forma, a principal contribuição deste trabalho é na linha de pesquisa de governança de sistemas multi-agentes abertos. A contribuição ocorre através da disponibilização de uma abordagem de governança com os objetivos de ser (i) flexível para dar suporte a evoluções na própria abordagem; (ii) permitir agregar e expressar os conceitos relacionados à governança encontrados na literatura atual; (iii) possuir um alto nível de abstração de seus elementos e ainda assim viabilizar o mapeamento da especificação para um mecanismo de implementação; e (iv) permitir a incorporação de aspectos de fidedignidade na especificação das leis. Uma vez desenvolvida a abordagem, mostra-se como a sua aplicação na construção de sistemas multi-agentes pode auxiliar a construção de sistemas que incorporam técnicas de fidedignidade.

1.1.

Problema

Garantir que o resultado observável resultante da interação entre os agentes de um sistema multi-agente aberto ocorra de acordo com o especificado e que o projetista do sistema possua ferramentas para lidar com atributos de fidedignidade. Mais especificamente, deve-se estabelecer instrumentos de governança que permitam que sistemas multi-agentes abertos adotem atributos de fidedignidade, considerando as seguintes hipóteses:

- (i) os detalhes da arquitetura e a implementação dos agentes são inacessíveis;
- (ii) o agentes interagem através de troca de mensagens;
- (iii) é possível especificar quais são as interações e comportamentos esperados do sistema a priori.

1.2.

Limitações de Abordagens Existentes

As abordagens atuais de governança que poderiam ser utilizadas para este propósito (i) dispõem de um conjunto restrito de entidades que permitem somente a especificação de leis em um nível de abstração muito baixo, contendo primitivas muito elementares; o que aumenta de forma significativa o esforço de desenvolvimento e

manutenção; (ii) ou são de um nível de abstração tão alto que não permitem o mapeamento automático da especificação para um mecanismo de implementação; (iii) ou possuem um conjunto de elementos adequados, porém baseado em um modelo inflexível que dificulta a adaptação da abordagem para evoluções tanto das leis quanto da abordagem em si.

1.3. Solução Proposta

Diante do problema especificado na Seção 1.1, nesta tese propõe-se uma abordagem de governança original para que leis de interação de um sistema multi-agente aberto possam ser especificadas, para que depois seja possível verificar se as interações estão seguindo a especificação e caso não estejam, ações corretivas possam ser tomadas. A abordagem proposta aborda as limitações identificadas na Seção 1.2 da seguinte forma:

Em relação ao item (i), parte dos elementos que compõem o meta-modelo são abstrações que fazem parte do estado da arte das pesquisas em governança [17]. Outros elementos foram introduzidos conforme a necessidade dos experimentos realizados e relatados em [18-25]. Isto permitiu a criação de um meta-modelo composto por entidades de alto nível de abstração e ainda assim com semântica suficientemente bem-definida para que seja possível especificar leis baseadas nestas abstrações e principalmente verificar em tempo de execução de um sistema se as leis estão sendo cumpridas, o que resolve o problema do item (ii).

Em relação a (iii), o meta-modelo é baseado em um modelo de eventos que permite fraco acoplamento entre as entidades, refletindo-se em flexibilidade para acomodar modificações e expressividade para compor os elementos do modelo. O mecanismo de eventos também é mapeado para o nível de implementação para um *middleware* baseado em componentes, no qual os componentes se comunicam principalmente através de eventos. Finalmente, em relação ao item (iv), este trabalho ilustra como duas técnicas de fidedignidade (tolerância a faltas e *dependability explicit computing*) podem ser aplicadas utilizando a abordagem proposta.

1.4. Contribuições

São contribuições desse trabalho²:

- (i) um meta-modelo de alto nível baseado em eventos para o domínio de leis de interação;
- (ii) uma notação gráfica que permite representar os elementos do meta-modelo;
- (iii) uma linguagem de representação declarativa que permite a especificação do meta-modelo. Esta linguagem é uma evolução do XMLaw [21, 26] e utiliza-se de analogias para os conceitos de programação por convenção (do inglês *coding by convention*) de projetos como Rails [27] e Grails [28] para aumentar a produtividade e a simplicidade das especificações das leis.
- (iv) um middleware capaz de monitorar interações em um sistema multi-agente e de interpretar as especificações das leis para verificar se elas estão sendo efetivamente cumpridas;
- (v) Ilustração da adoção de estratégias e conceitos da área de fidedignidade de forma integrada a abordagem proposta de utilização de leis de interação em sistemas multi-agentes abertos;
- (vi) Um estudo de caso realista que ilustra que a abordagem de leis pode ser aplicada para construir sistemas complexos mais confiáveis.

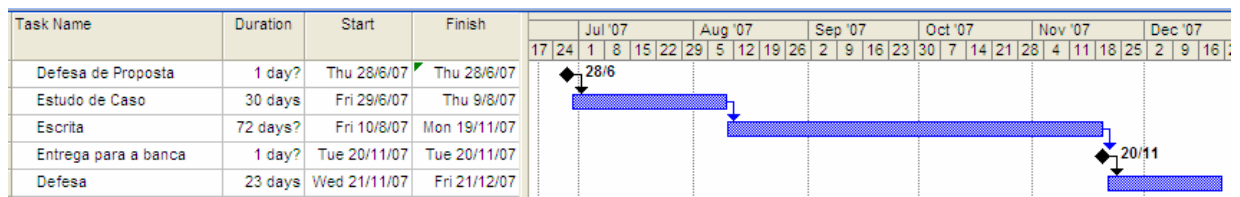
1.5. Organização da Tese

Capítulo 2 – O meta-modelo de leis: XMLaw	Neste capítulo, apresenta-se o meta-modelo de leis utilizado pela abordagem. O meta-modelo descreve todos os elementos de leis usados para se especificar uma lei. É apresentada a semântica, o relacionamento, uma notação gráfica e a forma de representação em uma linguagem declarativa de cada um dos elementos.
Capítulo 3- Trabalhos	Neste capítulo, apresentam-se as características

² Os itens de (i) a (iv) são referidos como a abordagem proposta.

relacionados ao meta-modelo	do meta-modelo do XMLaw que o diferencia dos modelos atuais. Discute-se a importância de se possuir elementos com alto nível de abstração e da flexibilidade para acomodar mudanças no modelo e na especificação.
Capítulo 4: Infra-estrutura de implementação: M-Law	Neste capítulo, apresenta-se um middleware baseado em um modelo de componentes que dá suporte ao monitoramento da interação entre os agentes com objetivo de verificar se as especificações das leis estão sendo seguidas.
Capítulo 5: Aplicando a abordagem de leis para alcançar fidedignidade (<i>dependability</i>)	Neste capítulo, é feito um levantamento bibliográfico que apresenta uma taxonomia que classifica os vários aspectos do conceito de fidedignidade e discute-se como a abordagem de governança pode contribuir para alcançá-los.
Capítulo 6: Estudo de Caso 1: implementando duas estratégias de tolerância a falhas através do XMLaw	Neste capítulo apresentam-se como implementar as estratégias <i>Full Forward Recovery</i> e <i>Partial Forward Recovery</i>
Capítulo 7: <i>Dependability Explicit Computing</i> e Leis	Neste capítulo, apresenta-se o conceito de <i>dependability explicit computing</i> . Apresenta-se uma implementação para ilustrar como realizar <i>dependability explicit computing</i> utilizando a abordagem proposta
Capítulo 8: Estudo de caso: controle de tráfego aéreo	Neste capítulo, apresenta-se um estudo de caso que ilustra a aplicação de todas as etapas do guia de desenvolvimento. Discutem-se também, como as leis estão ajudando a alcançar um maior grau de fidedignidade para este exemplo.
Capítulo 9: Conclusões e Trabalhos Futuros	Neste capítulo, são apresentadas reflexões sobre o trabalho realizado e discussões sobre pontos em aberto e trabalhos futuros.

1.6.Cronograma



2 Referências

1. Dignum, V., Dignum, F.: Modelling Agent Societies: Co-ordination Frameworks and Institutions. Proceedings of the 10th Portuguese Conference on Artificial Intelligence on Progress in Artificial Intelligence, Knowledge Extraction, Multi-agent Systems, Logic Programming and Constraint Solving. Springer-Verlag (2001)
2. Hannoun, M., Boissier, O., Sichman, J.S., Sayettat, C.: MOISE: An Organizational Model for Multi-agent Systems. Proceedings of the International Joint Conference, 7th Ibero-American Conference on AI: Advances in Artificial Intelligence. Springer-Verlag (2000)
3. DeLoach, S.A., Matson, E.: An Organizational Model for Designing Adaptive Multiagent Systems. The AAAI-04 Workshop on Agent Organizations: Theory and Practice (2004)
4. Ripper, P.S., Fontoura, M.F., Neto, A.M., Lucena, C.J.P.d.: V-Market: A framework for agent e-commerce systems. World Wide Web **3** (2000) 43--52
5. Guttman, R.H., Moukas, A.G., Maes, P.: Agent-mediated electronic commerce: a survey. Knowl. Eng. Rev. **13** (1998) 147-159
6. Michael, B., Alberto, S., James, A., Norman, S.: CMieux: adaptive strategies for competitive supply chain trading. SIGecom Exch. **6** (2006) 1-10
7. Lapinsky, S.E., Weshler, J., Mehta, S., Varkul, M., Hallett, D., Stewart, T.E.: Handheld computers in critical care. feedback (2004)
8. Esteva, M.: Electronic Institutions: from specification to development. IIIA PhD Monography **19** (2003)
9. Esteva, M., Rosell, B., Rodriguez-Aguilar, J.A., Arcos, J.L.: AMELI: An Agent-Based Middleware for Electronic Institutions. Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 1 (2004) 236-243
10. Dignum, F.: Abstract norms and electronic institutions. Proceedings of the International Workshop on Regulated Agent-Based Social Systems: Theories and Applications (RASTA'02), Bologna (2002) 93-104
11. Hannoun, M., Boissier, O., Sichman, J.S., Sayettat, C.: MOISE: An organizational model for multi-agent systems. Proceedings of the International Joint Conference, 7th Ibero-American Conference on AI, 15th Brazilian Symposium on AI (IBERAMIA/SBIA'2000), Atibaia, SP, Brazil, November (2000) 152-161
12. Naftaly, H.M., Victoria, U.: Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems. ACM Trans. Softw. Eng. Methodol. **9** (2000) 273-305

13. Noushin, A.: The impact of software process improvement on quality: in theory and practice. *Inf. Manage.* **40** (2003) 677-690
14. Kan, S.H.: *Metrics and Models in Software Quality Engineering*. Addison-Wesley Professional (2002)
15. Mahmood, N., David, W., Didar, Z.: A maturity model for the implementation of software process improvement: an empirical study. *J. Syst. Softw.* **74** (2005) 155-172
16. Lucena, C.J.P.d., Medeiros, C.B., Lucchesi, C.L., Maldonado, J.C., Almeida, V.A.F.: *Grandes Desafios da Pesquisa em Computação no Brasil – 2006 – 2016*. (2006)
17. Paes, R.d.B., Carvalho, G.R.d., Gatti, M., Lucena, C.J.P.d., Briot, J.-P., 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 III, Third International Workshop, E4MAS 2006, Hakodate, Japan, May 8, 2006, Selected Revised and Invited Papers*, Vol. 4389. Springer (2007) 221--238
18. Rodrigues, L.F., Carvalho, G., Paes, R., Lucena, C.: Towards an Integration Test Architecture for Open MAS. *Software Engineering for Agent-oriented Systems (SEAS 05)*, Uberlândia, Brazil (2005)
19. Carvalho, G., Paes, R., Lucena, C.: Extensions on Interaction Laws in Open Multi-Agent Systems. *Software Engineering for Agent-oriented Systems (SEAS 05)*, Uberlândia, Brazil (2005)
20. Carvalho, G., Paes, R., Lucena, C.: Governing the Interactions of an Agent-based Open Supply Chain Management System. *Monografias em Ciência da Computação n° 29/05*. Departamento de Informática, PUC-Rio (2005)
21. Paes, R.d.B.: *Regulando a Interação de Agentes em Sistemas Abertos - uma Abordagem de Leis*. Informática, Vol. Master. PUC-Rio, Rio de Janeiro (2005) 119
22. Gatti, M., Lucena, C.J.P.d., Briot, J.-P.: On Fault Tolerance in Law-Governed Multi-Agent Systems. *International Workshop on Software Engineering for Large-scale Multi-Agent Systems (SELMAS) at ICSE 2006, Shanghai, China* (2006)
23. Gatti, M., Paes, R., Carvalho, G., Rodrigues, L.F., Lucena, C.J.P.d., Faci, N., Briot, J.-P., Guessoum, Z.: Governing Agent Interaction in Open Multi-Agent Systems with Fault Tolerant Strategies. *International Workshop Agents and Multiagent Systems, from Theory to Application (AMTA'06), Quebec, Canada* (2006)
24. Carvalho, G.R.d., Almeida, H., Gatti, M., Ventura, G., Paes, R.d.B., Perkusich, A., Lucena, C.J.P.d.: Dynamic Law Evolution in Governance Mechanisms for Open Multi-Agent Systems. *Second Workshop on Software Engineering for Agent-oriented Systems (SEAS 2006)*, Florianópolis, Brasil (2006)
25. Gatti, M., Carvalho, G.R.d., Paes, R.d.B., Staa, A.v., Lucena, C.J.P.d., Briot, J.-P.: O Rationale da Fidedignidade em Sistemas Multiagentes Abertos Governados por Leis. *Second Workshop on Software Engineering for Agent-oriented Systems (SEAS 2006)*, Florianópolis, Brasil (2006)
26. Paes, R.d.B., Carvalho, G.R.d., Lucena, C.J.P.d., Alencar, P.S.C., Almeida, H., Silva, V.T.d.: Specifying Laws in Open Multi-Agent Systems. *Agents, Norms and Institutions for Regulated Multiagent Systems (ANIREM)*, Utrecht, The Netherlands (2005)

27. Thomas, D., Hansson, D.H., Breedt, L., Clark, M., Davidson, J.D., Gethland, J., Schwarz, A.: Agile Web Development with Rails. Programatic Bookshelf (2006)
28. Rocher, G.: The Definitive Guide to Grails (Definitive Guide). APRESS (2006)

ANEXOS

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 Cowan¹

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

¹University 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

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.

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 OMG³ 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.

³ Object Management Group – www.omg.org

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.

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.

...

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.

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.

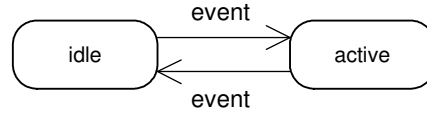


Fig. 2. Generic Law Element Lifecycle

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].

Relating the Model to Other Approaches

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 <code>imposeObligation</code> ,
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 <code>deliver([x,L'],m,y)</code> , delivers to the home actor the message m, ostensibly sent by x, operating under law L'.
Forward	Operation <code>forward(x,m,[y,L'])</code> sends the message m to Ty, the controller of the destination y—assumed 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 <code>incr(f,d)</code> , locates a unary term f(n), and increments its argument by d.
Decr	<code>decr(f,d)</code> is the implied counterpart of <code>incr</code> .
replaceCS	Operation <code>replaceCS(termList)</code> replaces the whole control-state of the home agent with the specified list of terms.
addCS	Operation <code>addCS(termList)</code> appends the terms in the list termList to the control state of the home agent.
imposeObligation	<code>imposeObligation(oType, dt, timeUnit)</code> 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 <code>repealObligation(oType)</code> removes all pending obligations of type oType, along with all associated obligation-terms in DCS.
imposeStateObligation	Operation <code>imposeStateObligation(termList)</code> would cause a <code>stateChanged</code> event to occur upon any change in any of the terms of the CS that are indicated by the termList parameter.
repealStateObligation	Operation <code>repealStateObligation(all)</code> 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.

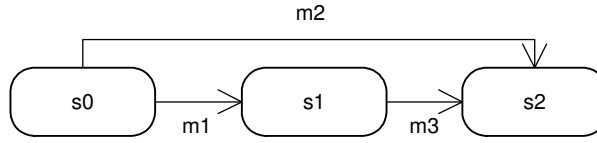


Fig. 3. Protocol Example

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 <i>m1</i> , a clock must be activated to fire an event in 5 seconds. The clock should be deactivated when the message <i>m2</i> arrives.
XMLaw myXMLawClock{5000,regular, (m1),(m2)}
LGI arrived(X, m1, Y) :- imposeObligation("myLGIclock",5). arrived(X, m2, Y) :- repealObligation("myLGIclock").
2. Fire transition <i>t2</i> when the clock generates a <i>clock_tick</i> event. The transition changes the protocol from state <i>s1</i> to state <i>s2</i> .
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 <i>m1</i> and deactivated by the arrival of message <i>m2</i> .
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 <i>m1</i> activates transition <i>t1</i> . The transition <i>t1</i> changes the protocol state from <i>s1</i> to <i>s2</i> . The norm <i>n1</i> must be activated when transition <i>t1</i> is fired. The norm is given to the agent that received the message <i>m1</i> .
XMLaw t1{s1->s2, m1}


```

nl{$addressee, (t1) }
LGI
sent(A,m1,Addressee) -> 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(nl,norm_activation) )
), do(add(norm(nl,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 nl is made active.

```

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.

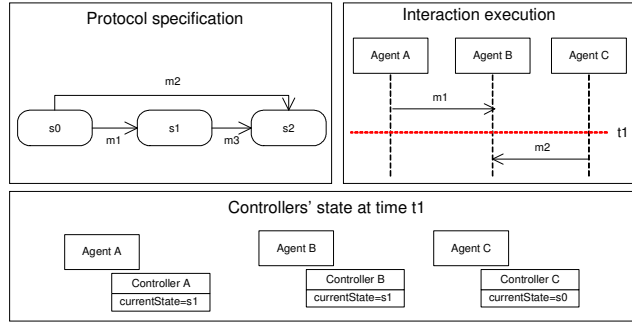


Fig. 4. Example of the need for synchronization of controllers to preserve global properties

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)).

```

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	Not	The Performative structure is a special type of scene

Structure	considered.	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 an 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.

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.

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*.

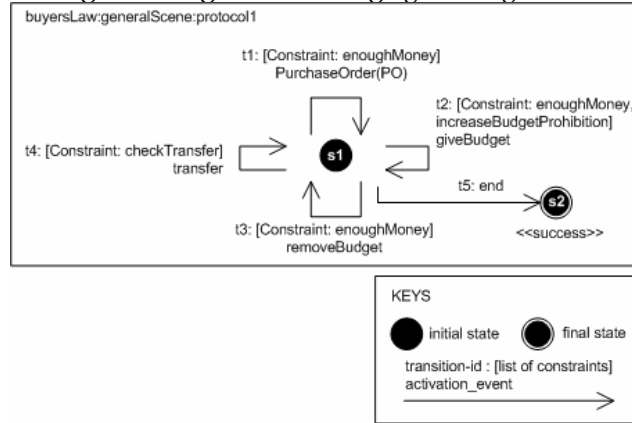


Fig. 5. Interaction Protocol

Table 11 – XMLaw Code

```

01: generalScene{
02:   PO{$budgetOwner, *, purchaseOrder($amount)}
03:   removeBudget(manager, $budgetOwner, removeBudget($amount))
04:   giveBudget{$budgetOwner, $receiver, giveBudget($amount)}
05:   transfer{$manager, $employee, transfer}
06:   end{$sender, $receiver, end}

07:   s1{initial}
08:   s2{success}

09:   t1{s1->s1, PO, [enoughMoney]}
10:   t2{s1->s1, giveBudget, [enoughMoney], [increaseBudgetProhibition]}
11:   t3{s1->s1, removeBudget, [enoughMoney]}
12:   t4{s1->s1, transfer, [checkTransfer]}
13:   t5{s1->s2, end}

14:   enoughMoney{br.pucrio.EnoughMoney}
15:   checkTransfer{br.pucrio.CheckTransfer}

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

19:   increaseBudgetProhibition{$budgetOwner, (t3), (t4)}
20:}
  
```

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.parseDouble(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.parseDouble(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.parseDouble(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.

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.

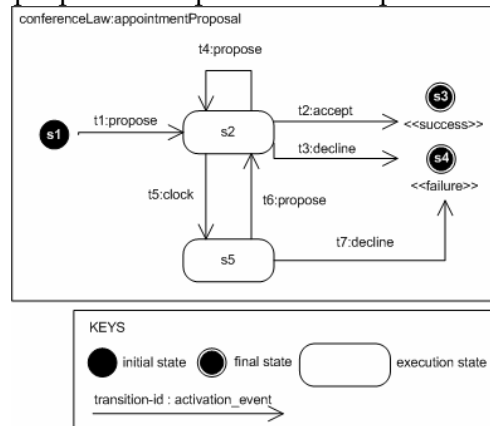


Fig. 6. Interaction Protocol

Table 16 - XMaw Code

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

05:   s1{initial}
06:   s3{success}
07:   s4{failure}

08:   t1{s1->s2, propose}
09:   t2{s2->s3, accept}
10:   t3{s2->s4, decline}
11:   t4{s2->s2, propose}
12:   t5{s2->s5, clock}
13:   t6{s5->s2, propose}
14:   t7{s5->s4, decline}

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

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

17:}
```

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.

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 multiagent 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 Computing Workshop 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>

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.

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

* 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

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.

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.

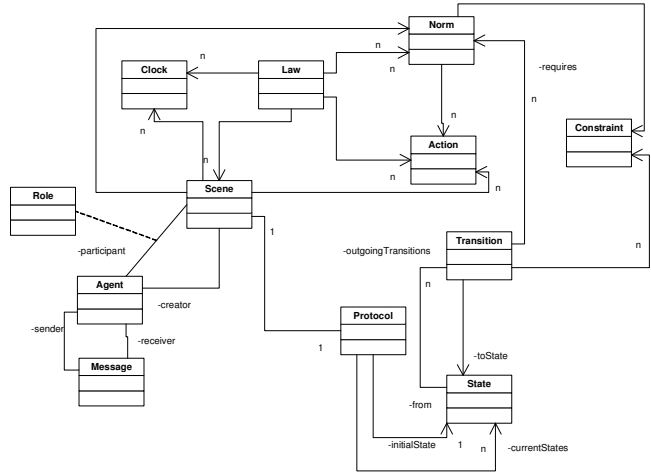


Fig. 1. XMLaw metamodel

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 t_1 , it may not be at time t_2 ($t_1 < t_2$). 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.

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 17**), Obligation and Prohibition elements are equal. Each type of norm contains activation and deactivation conditions. In **Table 17**, 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 17**, 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 17. XMLaw specification of the permission structure

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 18**) or Norm (**Table 17**) 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 18** 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 17**, 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.ValidateDate}
...
20:} // end scene
```

Table 18. Constraint *checkDueDate* used by a transition

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 17** 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.

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..

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.

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.

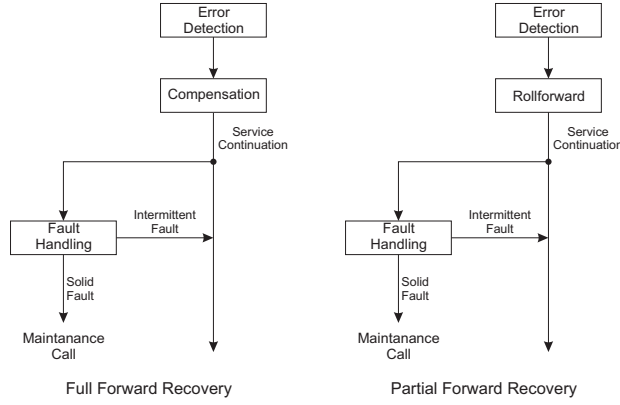


Fig. 2. Forward Recovery Strategy [8]

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.

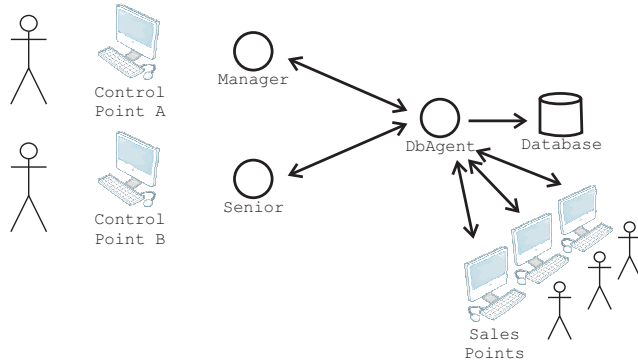


Fig. 3. Sales control system

The complete law specification is presented in **Table 19**. 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.

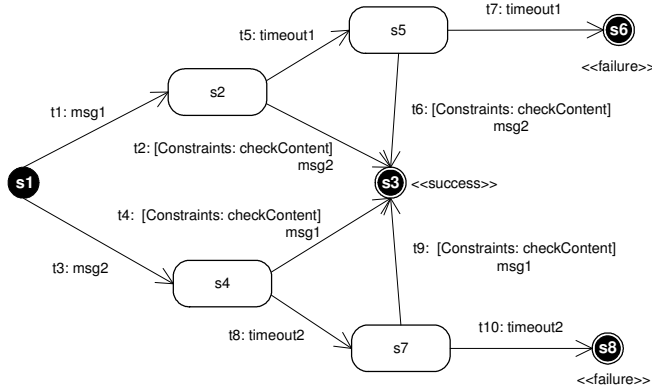


Fig. 4. Interaction Protocol

```
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 19. 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 *handleTimeout* 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.

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.

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. Berlin: 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.

Implementation of Dependability Explicit Computing in Open Multi-Agent Systems by Using Interaction Laws*

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

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

Abstract. In this paper we propose to incorporate the Dependability Explicit Computing (DepEx) ideas into a law-governed approach in order to build dependable open multi-agent systems. We show that the law specification can explicitly incorporate dependability concerns, collect data and publish them in a metadata registry. This data can be used to realize DepEx and, for example, it can help to guide design and runtime decisions. The advantages of using a law-approach are (i) the explicit specification of the dependability concerns; (ii) the automatic collection of the dependability metadata reusing the mediators' infrastructure presenting in law-governed approaches; and (iii) the ability to specify reactions to undesirable situations, thus preventing service failures.

Keywords: Interaction Laws, Dependability Explicit Computing, Multi-Agent Systems, Governance Mechanisms.

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

* 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

Introduction

Many current systems are open and dynamic. A key characteristic is that they demand dynamic binding, i.e. the selection and use of components, or agents, at run-time. Therefore, they do not consist simply of agents selected during an on-line design activity. Instead, they are open to agents arriving, departing or being modified. They are dynamic in order to provide services on a continuous basis, and do so even when agents or the environment change [1]. The greater the dependence of our society on such open distributed multi-agent systems, the greater will be the demand for dependable applications.

The dependability of a system can be defined as the ability to avoid service failures that are more frequent and more severe than is acceptable [2]. Dependability is an integrating concept that encompasses the following attributes [2]: (i) availability: readiness for correct service; (ii) reliability: continuity of correct service; (iii) safety: absence of catastrophic consequences on the user(s) and the environment; (iv) integrity: absence of improper system alterations; and (v) maintainability: ability to undergo modifications and repairs.

One way to promote dependability is by implementing a Dependability Explicit Computing (DepEx) approach [3]. DepEx treats dependability metadata as first-class data. The means for dependability (i.e., fault prevention, fault tolerance, fault forecasting and fault removal) should be explicitly incorporated in a development model focused on the production of dependable systems [3].

While developing the system, the data is specified by explicitly incorporating dependability-related information into system development from the earliest possible phases; by annotating design and implementation files with dependability-related metadata, which are updated when the files are processed (e.g. when development moves from one phase to another); and by maintaining this information to reflect the system and environment states afterwards.

Then, at the runtime or at the design time, the dependability metadata can be exploited to aid decision-making. Some examples of metadata are safety integrity level, failure rates, failure modes, pre and post conditions, MTBF, reliability, response time, resources consumed, component specification, fault assumptions, types of encryption, etc.

Achieving dependability in open multi-agent systems is particularly challenging. Such systems are characterized by having little or no control over the actions that agents can perform. Besides, the internal aspects of the agents (such as implementation language and architecture) are inaccessible. The research in interaction laws deals with this problem by explicitly specifying behavioral rules, and by providing mechanisms that check if the actual interactions conform to the specification at runtime. The mechanisms are usually implemented by either a central mediator [4] or by a decentralized community of mediators [5]. 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 propose to incorporate the Dependability Explicit Computing ideas into a law-governed approach in order to build dependable open multi-agent systems.

We show that the law specification can explicitly incorporate dependability concerns, collect data and publish them in a metadata registry. This data can be used to realize DepEx and, for example, it can help to guide design and runtime decisions. The advantages of using a law-approach are (i) the explicit specification of the dependability concerns; (ii) the automatic collection of the dependability metadata reusing the mediators' infrastructure presenting in law-governed approaches; and (iii) the ability to specify reactions to undesirable situations, therefore, preventing service failures.

This paper is organized as follows. In Section 0, we present a flexible law-governed approach called XMLaw. We use this approach throughout the examples given in this paper. In Section 0 we present in details of a case study where we discuss how the laws can be used to promote Dependability Explicit Computing. In Section 0, we specifically relate our research to previous work, explaining how the problem of promoting dependability in open multi-agent systems has been addressed so far. Finally, in Section 0, we present some discussions about this and future work.

XMLaw

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 [6][4] 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 [7][8]. 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 law specifications. A law specification is a description of law elements that are interrelated in a way that makes it 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 conceptual model of XMLaw is composed of the following main elements: {Event, Protocol, Transition, State, Scene, Clock, Norm, Constraint, Action}. The elements are described as follows.

- **Event** - an event models an occurrence related to the elements of the law. It can represent a change of state of a protocol, the arrival of a message sent by an agent, the moment in which an agent acquires an obligation, an announcement that a certain amount of time has elapsed and much more. The semantic of each event is determined by its type. There are many types of events, which are summarized as follows. Type of events = {message_arrival, compliant_message, transition_activation, failure_state_reached, successful_state_reached, failure_scene_completion, successful_scene_completion, scene_creation, time_to_live_elapsed, clock_activation, clock_tick, clock_timeout, clock_deactivation, norm_activation, norm_deactivation, norm_not_fulfilled, constraint_not_satisfied, action_activation}.

- **Protocol** - A protocol defines the possible states that an agent interaction can evolve. Transitions between states can be fired by any XMLaw event, instead of only message arrivals. Therefore, protocols specify the expected sequence of events in the path of interaction among the agents.
- **Transition** - a transition is a directed arc between a source state and an end state. It represents the change between two different situations in the course of the interactions caused by a response to the occurrence of an XMLaw event. Transitions may depend on norms and constraints to fire. If there is an obligation associated with the transition, then the obligation must be inactive in order to activate the transition. Instead, if there is a permission associated with the transition, the permission must be active in order to fire the transition. Constraints may act as fine-grained filters for transitions. A constraint could access a database, do some math, calculate date periods or perform any other complex domain-dependent operation in order to allow the transition to fire.
- **State** - A state models a possible step in the evolution of the agents' interaction. States can represent static or dynamic situations, such as *"waiting for buyer's answer"*, *"deciding about the deal"*, and so on. There are three types of states: successful, failure or execution. A successful state means that the protocol stops upon reaching success. A failure state means that the protocol stops with failure when the state is reached. For its part, when reached, an execution state does not stop the protocol.
- **Scene** - Scenes help organize interactions. The concept of scenes here is similar to those in theater plays, where actors play a role according to well defined scripts, and the entire play is composed of many connected scenes. A scene models an interaction context where protocols, actions, clocks and norms can be composed to represent complex normative situations. Furthermore, from the problem modeling point of view, a scene permits decomposing the problem into smaller and more manageable pieces of information. They can be viewed as building blocks of normative interactions. Normative interactions are situations in which agents interact through a set of behavioral rules, or social conventions.
- **Clock** - 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.
- **Norm** - A Norm [7][8] 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 structure of the Permission (Table 20), Obligation and Prohibition elements are equal. Each type of norm contains

activation and deactivation conditions. In **Table 20**, an assembler will receive the permission upon logging into 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 20**, the assembler agent (\$assembler) will receive the permission. Norms can also have constraints and actions associated with them. 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.

Table 20 - XMLaw specification of the permission structure

```
// 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
```

- **Constraint** - A constraint [7][8] is a restriction to 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 as complex constraints as needed for their applications. Constraints are defined inside Scene (**Table 21**) or Norm (**Table 20**) 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 21** 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 20**, 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.

Table 21 - Constraint checkDueDate used by a transition

```
01: negotiation{
```

```

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

```

- **Action** - An action supports the definition of the moment when the mediator should call a domain-specific service. 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. 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 shown in the example of **Table 20** 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.

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 law approaches [9][10]. The advantages claimed in favor of the use of events as a modeling element are also present in LGI [5], 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 study.

Grammar

Table 22 shows a simplified version of the XMLaw's grammar. The laws in XMLaw were originally written in an XML-based language [4][6][7][8]. That is the reason for the name XMLaw. However, the simplified notation presented here allows for a much clearer and compact specification of laws.

Table 22 – XMLaw Simplified Grammar

General Syntax:

| OR

[] optional

" reserved symbol

Message = message-id{'sender','addressee','content'}

Transition = transition-id{'sourceState'->'destinationState', message-id '}'
transition-id{'sourceState'->'destinationState', message-id ', Lists
'}

Lists = '[' list of constraints ids ']' |
[' list of norms ids ']' |
[' list of constraints ids ']' ',' [' list of norms ids ']

Clock = clock-id{' Time ',' Clock_Type ', ActivationEvents ', DeactivationEvents '}

Time: IntegerLiteral[Unit]

Unit: 's' | 'm' | 'h' | 'd'

Clock_Type = 'periodic' | 'regular'

ActivationEvents = Events

DeactivationEvents = Events

Events = '(' |
'(ElementRef)' |
'(ListsOfElementsRef)' |
'(element-id..'element-id)'

ElementRef = element-id |
'(element-id ', event-type ')

Contraint = constraint-id{'java-class'}

```
Action = action-id{'ActivationEvents ','java-class'}
```

```
Norm = norm-id{' NormType ',' owner ',' ActivationEvents ',' DeactivationEvents '}
```

```
NormType = 'obligation' | 'permission' | 'prohibition'
```

Implementing DepExp Using XMLaw

In this section, we present a motivating case study to illustrate how to specify the laws in such a way that the dependability metadata is treated as first-class data. The problem description was already reported in [11], and it was slightly modified to this case study.

Problem Description

Consider the task of creating a system composed of three types of agents: a travel agent, a user agent and an airline agent. The airline provides several related operations, which must be invoked according to a complex conversation protocol. Assume that the airline agent provides five different operations: *checkSeatAvailability*, *reserveSeats*, *cancelReservation*, *bookSeats*, and *notifyExpiration*. Each operation performs a single airline travel related task. The operations must be invoked by a client according to the following conversation rules:

- *checkSeatsAvailability* must be the first operation to be invoked;
- *reserveSeats* may only be invoked if a client has already invoked *checkSeatsAvailability* and the requested seats are indeed available; the reservation is held only for a certain amount of time;
- *bookSeats* or *cancelReservation* may be invoked, but only if the seats have been reserved (by a successful invocation of *reserveSeats*) and the reservation has not expired;
- if neither *bookSeats* nor *cancelReservation* has been invoked by the client within a specified amount of time, the airline agent will itself invoke *notifyExpiration* to inform the client that the reservation has expired.

A traveler, represented by the user agent, planning on taking a trip submits a *TripOrder* (through *getItinerary* message) to her travel agent, hoping to get an *Itinerary* proposal. The *TripOrder* contains information such as departure and destination, departure date and time, and return date and time (for a round trip), the number of maximum connections and the number of travelers.

The travel agents finds the best itinerary to reach the destination based on the traveler's criteria such as the cheapest fare, availability of flights, or frequent flyer miles

accumulated by the traveler. Before the *Itinerary* can be proposed to the traveler, the travel agent invokes the airline agent to verify the availability of seats (*checkSeatsAvailability*). In the event the seats are available, the travel agent notifies the traveler and waits for the traveler to submit a modified *TripOrder*.

If seats are available, the proposed *Itinerary* is sent to the traveler for confirmation. She then decides to reserve the seats for the *Itinerary* and gives the travel agent her contact information so that the airline agent will be able to send her an e-Ticket.

Next, the travel agent interacts with the airline agent to electronically finalize the reservation (*reserveSeats*). Let us assume that the airline holds such reservation for one day, and that if a *BookRequest* is not received within one day, the seats are released and the travel agent is notified. The travel agent sends a *ReserveResult* message to the traveler as an acknowledgment.

At this point, the traveler can either book or cancel the reservation. If she decides to book the trip, she sends a *BookRequest* to the travel agent containing her credit card information. The travel agent then invokes the *bookSeats* operation of the airline agent to finally book the seats. As a result, the airline agent books the seats for the proposed *Itinerary*, and issues an e-ticket to the traveler.

Architecture

The architecture of the system is shown in **Figure 7**. The architecture is based on the metadata architectural model presented in [1]. The architecture was conceived for the achievement of predictable levels of dynamic resilience in distributed systems. We have chosen this architecture because it already contains the components to enable DepExp. It has a metadata registry, a runtime reasoning/adaptation service and a metadata acquisition component.

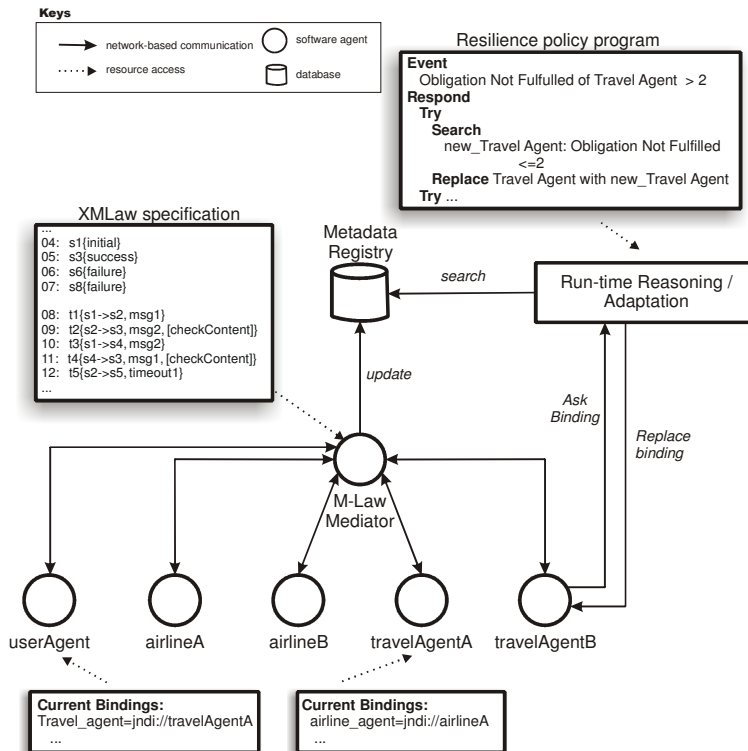


Figure 7- System architecture

In this case study, the role of the metadata acquisition component is performed by the M-Law middleware. M-Law works by intercepting messages exchanged between agents, verifying the compliance of the messages with the laws and subsequently redirecting the message to the real addressee, if the laws allow it. If the message is not compliant, then the mediator blocks the message and applies the consequences specified in the law (**Figure 8**). This architecture is based on a pool of mediators that intercept messages and interpret the previously described laws. A more detailed explanation about how this architecture was in fact implemented can be found in [4]. As more clients are added to the system, additional mediators' instances can be added to improve throughput.

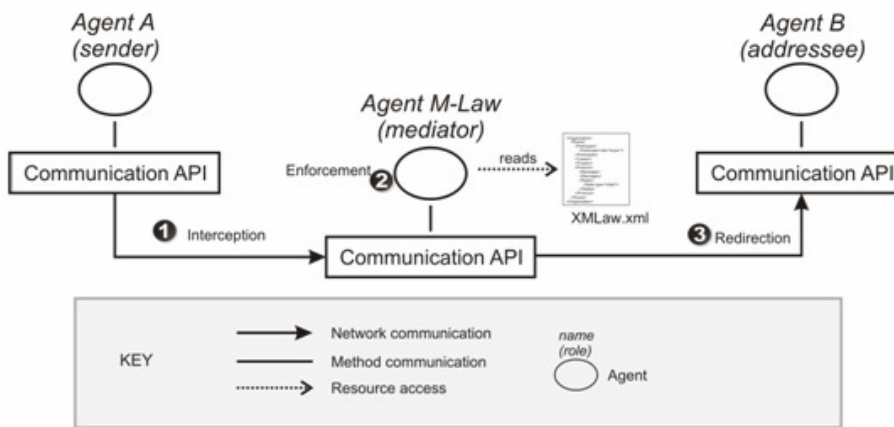


Figure 8 - M-Law architecture

The M-Law mediates the communication between the agents. The behavior of the M-Law is specified in the XMLaw file that it reads. In the XMLaw specification there are instructions that tell the mediator how to update the metadata registry. At runtime, the metadata registry can be used in two ways: directly by the agents or through the runtime reasoning. The agents can proactively search for metadata and self-adapt to reflect their dependability requirements. As an example, a user agent can search for a travel agent that has not broken any obligation during the last month.

In this case study, there are two available travel agents (*travelAgentA* and *travelAgentB*). At runtime, the user agent is able to choose which travel agent she will interact with based on the dependability metadata. The travel agents have two available airline agents with which they can interact. The choice of which of them is used can be also based on the dependability metadata available at the metadata registry

The runtime reasoning provides different tasks related to the processing of metadata stored in the metadata registry, such as comparison/matching of metadata, determination of equivalent metadata information and composition of metadata [1]. This service manages the list of agents, seamlessly activating or connecting the ones that will be used according to a specified resilience policy program. As an example, in the **Figure 7**, the *travelAgentB* is interacting with the runtime reasoning. In this paper, we focus on how we can specify the laws to automatically update the metadata registry.

The meta-data

We are using the laws to specify metadata concerning availability, service failure and enforcement of pre and post conditions.

- **Availability** – every time an agent sends a request to other agent, the receiver should answer within a pre-specified amount of time. The absence of an answer implicates that at that time the receiver is not available with the required quality level.
- **Service failure** – during the interaction, agents acquire obligations that they must fulfill. The fulfillment of these obligations represents the expected correct behavior for the agents. Therefore, each obligation that is not fulfilled can be interpreted as a service failure, i.e., the actual system execution deviates from the correct behavior.
- **Pre and post conditions** - Agent specifications may likely change as agents evolve, but resilience may be maintained if it is possible to reason dynamically about the functional properties of agents, including abnormal behaviors. Dynamic resilience mechanisms require agent specifications as metadata, including both specifications of services offered and services required. These can be given by pre-/post-conditions, potential failure behaviors and responses when the components are used outside their pre-conditions. As an example of pre-condition, let us suppose that we want to enforce that the values of the departure and destination attributes specified in the *TripOrder* (*getItinerary* message) must belong to the set of possible attributes $S=\{“Toronto”, “New York”, “London”, “Tokyo”, “Rio de Janeiro”\}$. Enforcing this constraint guarantees that the travel agent is receiving a parameter that is within its specification scope.

Therefore, considering the case study description (Section 0) and the metadata concerns described above, it is possible to specify the following law requirements:

- **Requirement #1** - The whole process must occur within two days. After two days, the process is cancelled and all the rules are no longer valid. All the interactions must restart.
- **Requirement #2** – All interactions must occur as the pre-defined order specified in the problem description (Section 0).
- **Requirement #3** - If the airline agent says that there is a seat available, this seat must be saved to the travel agent for at least five minutes. This way, the user has some time for deciding about the confirmation of the reservation. If the time has elapsed, and the airline has not received any confirmation, then the airline is allowed to answer with a *not-available* message and book the seat for another client.
- **Requirement #4** - When the airline agent sends a *result-ok* message in response to a seat reservation, then the reservation must be held for at least one day

- **Requirement #5** - The *TripOrder* (*getItinerary* message) must belong to the set of possible attributes $S=\{“Toronto”, “New York”, “London”, “Tokyo”, “Rio de Janeiro”\}$.
- **Requirement #6** - Every request that does not require user interaction must be answered within 15 seconds by any agent.

Metadata acquisition through XMLaw specification

The interaction protocol is shown in **Figure 9** and the complete XMLaw specification can be found in **Table 24**. The scene is declared in lines 01 and 02. Lines 03 to 16 contain the pattern of messages that agents are expected to exchange. Lines 17 to 20 specify the initial and final states of the interaction protocol. The transitions are specified in lines 21 to 37. The transitions refer to the states, messages, constraints and norms present in the law. Clocks are specified in lines 38 to 40, constraint in line 41, actions in lines 42 to 44, and norms in lines 45 and 46. Next, we show how the six law requirements were specified in the laws.

- Requirement #1: This requirement is implemented as the *time-to-live* scene attribute in line 02.
- Requirement #2: The interaction protocol in **Figure 9** reflects exactly the possible paths of interactions described for the case study. This protocol is specified in lines 03 to 37. These lines declare all messages, states and transitions present in the protocol.
- Requirement #3: This requirement demands a combined use of various XMLaw elements. First, it is necessary to identify when the airline agent “says there is a seat available”. Then, we have to start to count five minutes. The airline is not allowed to answer *not-available* within these five minutes. **Table 23** shows the sequence of observed events that makes it possible to specify this requirement. This table is mapped to the XMLaw specifications in lines 35, 39, 43 and 45 (highlighted in **Table 24**).

Table 23 – Rationale for the XMLaw specification of the requirement #3 in lines 35, 39, 43 and 45.

Airline agent sends <i>itinerary-1</i> message to the travel agent. It means that airline agent is saying: “there is a seat available”. Then, we activate the clock to start counting the time. Moreover, we also activate the obligation <i>hold-seat</i> , and give it to the airline agent.
WHEN (t3, transition_activation)
ACTIVATE hold-seat-clock, hold-seat
If the time that the airline must hold the seat has elapsed (<i>clock_tick</i> event), then the obligation does not need to be fulfilled, i.e., the airline agent can answer with a <i>not-available</i> message.
WHEN (hold-seat-clock, clock_tick)
DEACTIVATE hold-seat
If the airline agent answers with a <i>result-ok</i> to a <i>reserveSeats</i> requisition, it means that the

airline agent has fulfilled its obligation of holding the seat. Then, the obligation should be deactivated.
WHEN (t7, transition_activation) DEACTIVATE hold-seat
The transition <i>t15</i> only fires if the obligation <i>hold-seat</i> is deactivated. If the airline sends the <i>not-available</i> message while the obligation is still active, then a <i>norm_not_fulfilled</i> event will be generated and the transition will not fire. As we are concerned with acquiring metadata about agents that do not follow the rules, the non-fulfillment of an obligation should be reported to the metadata registry. The action <i>updateHoldSeatMetadata</i> is in charge of obtaining the contextual information such as the agent, the obligation id (in this case <i>hold-seat</i>) and updating the registry.
WHEN (hold-seat, norm_not_fulfilled) ACTIVATE updateHoldSeatMetadata

- Requirement #4 – This requirement is specified in XMLaw using an idea similar to requirement #3. The transition *t16* (line 36) only fires if the obligation *hold-reservation* (line 46) is deactivated. The clock *hold-reservation-clock* (line 40) counts the time until one day. And the action *updateHoldReservationMetadata* updates the metadata registry with information about agents that do not fulfill the obligation.
- Requirement #5 – The constraint *checkContent* specified in line 41 is invoked by the transition *t1* (line 21). The constraint verifies if the values of the variables *dep* and *dest* (line 03) belong to the set of the pre-defined cities. The constraint implementation is shown in **Table 25**.
- Requirement #6 – This requirement states that agents that do not wait for input from the user must not take too long to provide an answer. The *availability-clock* specified in line 38 counts 15 seconds every time an agent receives a request. The clock is reset when the agent answers the request. The transitions *t1, t2, t3, t5, t6, t7, t9, t10, t11*, and *t13* specified in the clock, represent requests to agents. Note that transitions such as *t4* are not present in this list. This is because *t4* represents a message that is sent to the user (through the user agent). Every time an agent does not answer within the 15s, the clock generates a *clock_tick* event. This event is listened to by the action *updateClockMetadata* (line 42). The action updates the metadata registry indicating that the agent was not available at that time. The code for this action is shown in **Table 15**.

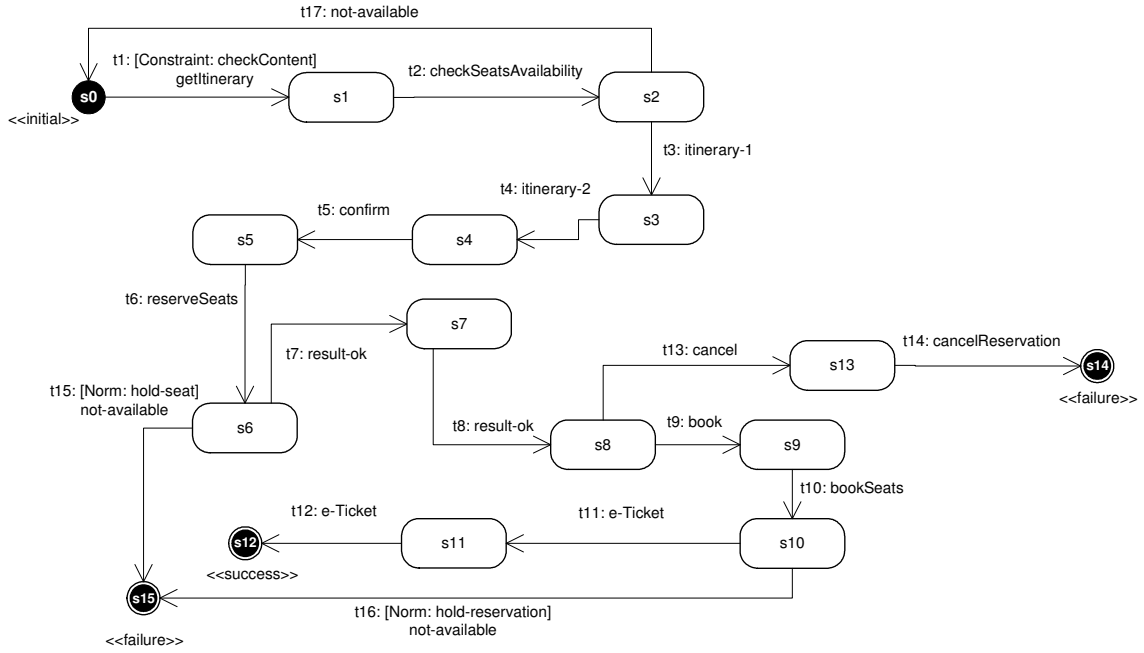


Figure 9 – Protocol Specification

Table 24 – XMLaw specification

// scene specification	
01:	planningATrip{
02:	time-to-live=2d
// pattern of messages	
03:	getItinerary{userAgent,travelAgent,trip_order(\$dep, \$dest, \$depDate, \$depTime, \$retDate, \$retTime, \$maxCon, \$travellers)}
04:	checkSeatsAvailability{travelAgent,airlineAgent, \$trip_order}
05:	itinerary-1{airlineAgent, travelAgent, itinerary(\$id,\$details)}
06:	itinerary-2{travelAgent, userAgent, itinerary(\$id,\$details)}
07:	confirm{userAgent, travelAgent, confirm(\$id)}
08:	reserveSeats{travelAgent, airlineAgent, reserveSeats(\$id)}
09:	result-ok-1{airlineAgent, travelAgent, ok(\$id)}
10:	result-ok-2{travelAgent, userAgent, ok(\$id)}
11:	book{userAgent, travelAgent, book(\$id)}
12:	bookSeats{travelAgent, airlineAgent, bookSeats(\$id)}
13:	e-Ticket{\$sender, \$receiver, e-ticket(\$ticketId)}
14:	cancel{userAgent, travelAgent, cancel(\$id)}

```

15: cancelReservation{travelAgent, airlineAgent, cancelReservation($id)}
16: not-available{airlineAgent, travelAgent, not-available($id)}

// initial and final states
17: s0{initial}
18: s12{success}
19: s14{failure}
20: s15{failure}

// transitions
21: t1{s0->s1, getItinerary, [checkContent]}
22: t2{s1->s2, checkSeatsAvailability}
23: t3{s2->s3, itinerary-1}
24: t4{s3->s4, itinerary-2}
25: t5{s4->s5, confirm}
26: t6{s5->s6, reserveSeats}
27: t7{s6->s7, result-ok-1}
28: t8{s7->s8, result-ok-2}
29: t9{s8->s9, book}
30: t10{s9->s10, bookSeats}
31: t11{s10->s11, e-Ticket}
32: t12{s11->s12, e-Ticket}
33: t13{s8->s13, cancel}
34: t14{s13->s14, cancelReservation}
35: t15{s6->s15, not-available, [hold-seat]}
36: t16{s10->s15, not-available, [hold-reservation]}
37: t17{s2->s0, not-available}

// Clocks
38:          availability-clock{15s,      regular,      (t1,t2,t3,t5,t6,t7,t9,t10,t11,t13),
(t2,t3,t4,t6,t7,t8,t10,t11,t12,t16,t17)}

```

```

39: hold-seat-clock{5m, regular, (t3), (t6)}
40: hold-reservation-clock{1d, regular, (t7), (t10)}

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

// Actions
42: updateClockMetadata{(availability-clock), br.pucrio.DecAvailability}
43: updateHoldSeatMetadata{((hold-seat, norm_not_fulfilled)), br.pucrio.HoldSeat}
44:      updateHoldReservationMetadata{((hold-reservation,  norm_not_fulfilled)),
br.pucrio.HoldReservation}

// Norms
45: hold-seat{obligation, airlineAgent, (t3), (hold-seat-clock, t7)}
46: hold-reservation{obligation, airlineAgent, (t7), (hold-reservation-clock , t11)}

47:}

```

Table 25 – Java Implementation of the CheckContent Constraint

```

class CheckContent implements IConstraint{
    private static List<String> allowed = new ArrayList<String>();
    private void init(){
        allowed.add("Toronto");
        allowed.add("New York");
        allowed.add("London");
        allowed.add("Tokyo");
        allowed.add("Rio de Janeiro");
    }
    public boolean constrain(ReadOnlyContext ctx){
        String dep = ctx.get("dep");
        String dest = ctx.get("dest");
        if ( !allowed.contains(dep) || !allowed.contains(dest) ){

```

```

        return true; // constrains, transition should not fire
    }
}
}

```

Table 26 - Action *updateClockMetadata* implemented as the java class *DecAvailability*

```

class DecAvailability implements IAction{
    private Datasource metadataRegistry;
    ...
    public void execute(Context ctx){
        String addressee = ctx.get("lastAddressee");
        Event event      = ctx.get("activationEvent");
        metadataRegistry.insert(event, addressee);
    }
}

```

The Metadata registry

In this case study, the metadata registry is composed of two entities: *agent* and *dependability_data*. The entity-relationship model is presented in **Figure 10**, and it is described in **Table 27**.

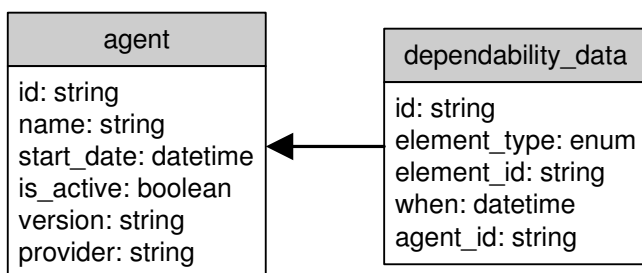


Figure 10 - Entity-relationship model of the metadata registry

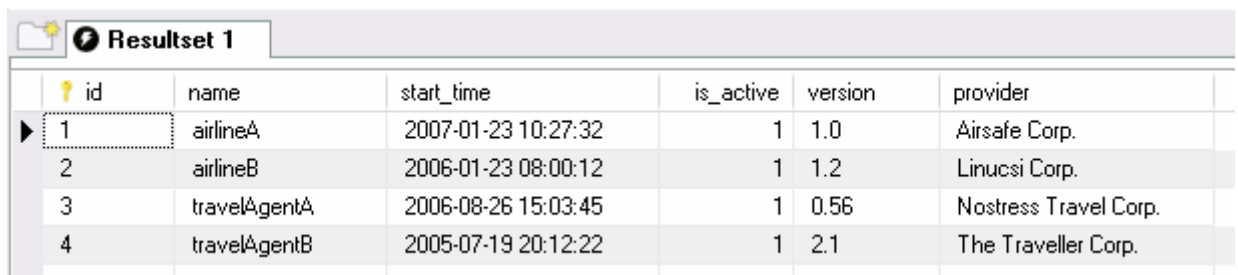
Table 27 - Description of the Attributes

agent	dependability_data
id - unique database identifier	id - unique database identifier of

of the agent.	the data.
name - the unique name of the agent	element_type - type of the XMLaw element. (eg: obligation, clock, ...)
start_date - date when the agent was added to the registry	element_id - id if the XMLaw element
is_active - true if the agent is still running	when - date and time of the insertion of the metadata
version - version of the agent	agent_id - agent identification associated with this metadata
provider - organization that is in charge of the agent	

The actions *updateClockMetadata*, *updateHoldSeatMetadata*, *updateHoldReservationMetadata* (lines 42, 43 and 44) are responsible for updating the metadata registry. In fact, these actions update the dependability metadata of the agents at runtime. **Figure 12** and **Figure 11** show screenshots of the registry database. For example, **Figure 12** shows that the *airlineA* (agent_id=1) has not fulfilled the obligations *hold-seat* at February 1st and *hold-reservation* at February 3rd.

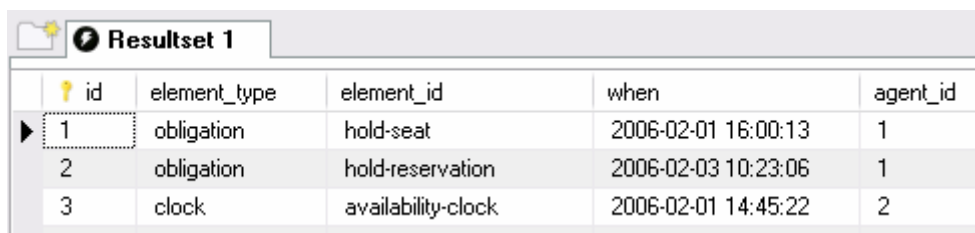
It is important to notice that these actions are very simple elements that obtain the contextual information at runtime and update the metadata registry. However, it is the law specification that tells when the actions should execute. In other words, the acquisition of the dependability metadata is done through the combined use of various XMLaw elements. It can clearly be seen in **Table 23**, where a transition, a clock, a norm and an action were connected to update the metadata.



The screenshot shows a table titled 'Resultset 1' with the following data:

id	name	start_time	is_active	version	provider
1	airlineA	2007-01-23 10:27:32	1	1.0	Airsafe Corp.
2	airlineB	2006-01-23 08:00:12	1	1.2	Linucsi Corp.
3	travelAgentA	2006-08-26 15:03:45	1	0.56	Nostress Travel Corp.
4	travelAgentB	2005-07-19 20:12:22	1	2.1	The Traveller Corp.

Figure 11 - Examples of the agent



The screenshot shows a table titled 'Resultset 1' with the following data:

id	element_type	element_id	when	agent_id
1	obligation	hold-seat	2006-02-01 16:00:13	1
2	obligation	hold-reservation	2006-02-03 10:23:06	1
3	clock	availability-clock	2006-02-01 14:45:22	2

Figure 12 - Examples of the dependability_data.

Surely, complex queries can be built using this simple model. For example, to query the number of not fulfilled obligations of the *airlineA*, one can write an SQL command such as follows.

```
SELECT count(id) as "Obligation Not Fulfilled" FROM dependability_data WHERE  
element_type='obligation' and agent_id='1'
```

Case Study Discussions

The case study presented has three main contributions: (i) first it shows the integration of a law mechanism (M-Law) into the architecture that allows for dynamically resilient systems based on a metadata approach; (ii) second it shows that the laws can be a very powerful way not only to acquire dependability metadata, but also to interfere in the system execution when necessary (iii) Finally, with the laws, the dependability concerns are explicitly considered and precisely specified mostly in a declarative way.

Related Work

In [12], a position paper is presented that illustrates the use of OWL for dependability specifications. An advantage claimed by the author is that the ontology includes the definition of a controlled vocabulary. With the potential of confusion between parties, and domains about the meaning of dependability metrics, an ontology is therefore valuable for disambiguation.

In [13], a tool was presented for monitoring the dependability and performance of Web Services. The metadata acquisition and maintenance occurs from a specific location of a client (reflects problems with routers and service WSs). The results are collected and updated in an openly accessible DB. The tool measures the dependability of Web Services by acting as a client to the Web Service under investigation. The tool monitors a given Web Service by tracking the following reliability characteristics: (i) availability: the tool periodically makes dummy calls to the Web Service to check whether it is running; (ii) functionality: the tool makes calls to the Web Service and checks the returned results to ensure the Web Service is functioning properly; (iii) performance: the tool monitors the round-trip time of a call to the Web Services producing and displaying real time statistics on service performance; (iv) and faults and exceptions: the tool logs faults and exceptions during the test period of the Web Service for further analysis. Although the tool can be useful for many existent applications, when compared to the solution presented in this paper, the laws allow for a much more expressive and flexible way to collect domain-specific situations. For example, in the tool it is not possible to express any of the obligations stated in our case study.

Furthermore, to the best of our knowledge there is no solution that encompasses the various features presented in this paper: (i) enforcement of the interaction behavior; (ii) flexible and declarative behavior of the interactions; (iii) explicit incorporation of dependability concerns into the specification; (iv) and openly accessible database about dependability metadata (metadata registry).

Discussions

Our society is becoming increasingly dependent on complex software systems. This dependence in turn makes the task of building dependable systems a critical part of software development. Dependability explicit computing states that there is a need for integrating the dependability concerns at the very early stages of the development process. In this approach the dependability metadata should be specified as first-class entities that are available to guide decisions both at the design time and at run-time.

On the other hand, the law-governed approaches have already proposed various high level elements that allow for a flexible specification of the overall system behavior. Furthermore, they provide mediators that ensure that the behavior is being followed as expected. Law-governed approaches also promote dependability in the sense that the system becomes more predictable and some system failures can be prevented by the intervention of the mediator.

In this paper, we have shown that DepEx and Law approaches are complementary. The laws can provide a powerful way to monitor and specify complex dependability metadata. To be more specific, we have incorporated the dependability explicit computing into the XMLaw approach. A detailed case study was presented with the goal of illustrating the acquisition of the metadata. We have also shown the model of a metadata registry and how this model can be queried to return dependability metadata.

The approach presented in this paper has the advantage of flexibility and reuse. Flexibility, because in contrast to the related work, the high level abstractions presented in XMLaw allow for a very expressive and domain dependent way to acquire the metadata, while still preserving the declarative nature of the laws. And reuse, because we do not have to rebuild a new language nor a new mediator to perform the metadata acquisition. Therefore, XMLaw was shown to be flexible enough to incorporate various dependability concerns.

Acknowledgements

The authors would like to thank Dr. Alexander Romanovsky for indicating the possibility of using our law-governed approach in the dependability domain.

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]. Di Marzo Serugendo, G., Fitzgerald, J., Romanovsky, A. and Guelfi, N., Dependable Self-Organising Software Architectures - An Approach for Self-Managing Systems, Technical Report No: BBKCS-06-05, School of Computer Science and Information Systems, Birkbeck College, London, May 2006
- [2]. Avizienis, A., Laprie, J-C., Randell, B., and Landwehr, C., Basic Concepts and Taxonomy of Dependable and Secure Computing, IEEE Transactions on Dependable and Secure Computing, vol. 1, n. 1, pp. 11-33, January-March, 2004

- [3]. Kaâniche, M., Laprie, J., and Blanquart, J. 2000. A Dependability-Explicit Model for the Development of Computing Systems. In Proceedings of the 19th International Conference on Computer Safety, Reliability and Security (October 24 - 27, 2000). F. Koornneef and M. v. Meulen, Eds. Lecture Notes In Computer Science, vol. 1943. Springer-Verlag, London, 107-116.
- [4]. R. Paes, M. Gatti, G. Carvalho, L. Rodrigues, C. Lucena, A middleware for governance in open multi-agent systems, Tech. Rep. MCC 33/06, PUC-Rio, <http://wiki.les.inf.puc-rio.br/uploads/8/87/Mlaw-mcc-agosto-06.pdf> (2006).
- [5]. N. H. Minsky, V. Ungureanu, Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems, ACM Trans. Softw. Eng. Methodol. 9 (3) (2000) 273--305.
- [6]. R. Paes, G. Carvalho, M. Gatti, C. Lucena, J.-P. Briot, R. Choren, Enhancing the Environment with a Law-Governed Service for Monitoring and Enforcing Behavior in Open Multi-Agent Systems, Vol. to appear of Lecture Notes in Computer Science, Springer, 2007.
- [7]. R. Paes, G. Carvalho, C. Lucena, P. Alencar, H. Almeida, V. Silva, Specifying laws in open multi-agent systems, in: Agents, Norms and Institutions for Regulated Multiagent Systems - ANIREM, Utrecht, The Netherlands, 2005.
- [8]. R. Paes, G. Carvalho, C. Lucena, XMLaw specification: version 1.0, Tech. Rep. to appear, PUC-Rio, Rio de Janeiro, Brasil (2007).
- [9]. M. Esteva, Electronic institutions: from specification to development, Ph.D. thesis, Institut d'Investigaci en Intel.ligència Artificial, Catalonia - Spain (October 2003).
- [10]. V. Dignum, J. Vázquez-Salceda, 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.
- [11]. X. Yi and K. Kochut, "Process Composition of Web Services with Complex Conversation Protocols: a Colored Petri Nets Based Approach", Proc. of Design, Analysis, and Simulation of Dist. Sys. Symposium, 2004
- [12]. Dobson, G., (2006) "OWL and OWL-S for Dependability-Explicit Service-Centric Computing", Service-Oriented Computing: Consequences for Engineering Requirements (SOCCER'06 - RE'06 Workshop), p. 4, September 2006
- [13]. Y. Chen, P. Li, A. Romanovsky. Web Services Dependability and Performance Monitoring. Proc. of 21st UK Performance Engineering Workshop. Newcastle upon Tyne. UK. July 2005.