



Rodrigo de Barros Paes

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

Tese de Doutorado

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, 02 de Outubro de 2007

Todos os direitos reservados. É proibida a reprodução total ou parcial do trabalho sem autorização da universidade, do autor e dos orientadores.

Rodrigo de Barros Paes

Mestre em Informática pela PUC-Rio em 2005. Graduou-se em Ciência da Computação na Universidade Federal de Alagoas em 2003. É pesquisador associado ao Laboratório de Engenharia de Software (LES) da PUC-Rio, atuando na área de Engenharia de Software de Sistemas Multi-agentes e Qualidade de Software.

Paes, Rodrigo de Barros

Governança de Sistemas Multi-Agentes Abertos com Fidedignidade / Rodrigo de Barros Paes; orientador: Carlos José Pereira de Lucena, – Rio de Janeiro: PUC, Departamento de Informática, 2007.

197 f. . : il. (col.) ; 30 cm

Tese (Doutorado em Informática)–Pontifícia Universidade Católica do Rio de Janeiro, Rio de Janeiro, 2007.

Inclui bibliografia

1. Informática – Teses. 2. Sistemas Multi-agentes 3. Sistemas Abertos 4. Governança baseada em Leis 5. Leis de Interação 6. Fidedignidade

CDD: 004



Rodrigo de Barros Paes

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

Tese apresentada ao Programa de Pós-graduação em Informática da PUC-Rio como requisito parcial para obtenção do grau de Doutor em Informática. Aprovada pela Comissão Examinadora abaixo assinada.

Prof. Carlos José Pereira de Lucena

Orientador

Departamento de Informática – PUC-Rio

Prof. Arndt von Staa

Departamento de Informática – PUC-Rio

Profª. Simone Diniz Junqueira Barbosa

Departamento de Informática – PUC-Rio

Prof. Jaime Simão Sichman

Departamento de Engenharia de Computação e Sistemas Digitais - Escola Politécnica da Universidade de São Paulo

Prof. José Carlos Maldonado

Instituto de Ciências Matemáticas e de Computação – Universidade de São Paulo – São Carlos

Prof. José Eugenio Leal

Coordenador Setorial do Centro Técnico Científico – PUC-Rio

Rio de Janeiro, 02 de Outubro de 2007

"Tentar não significa conseguir. Mas, certamente, todos os que conseguiram tentaram..."

Agradecimentos

“Ô mô fio quando é que você vai começar a trabalhar? Só quer saber de estudar né? Ô vida mansa! E a tese, defende quando? Pense num bixo nerd! Fique aí estudando que eu vou pegar uma prainha!”

Pois é, durante alguns anos escutei muitas variações das frases acima. Agora terminou mais uma etapa, uma etapa importante. Enfim! Agora o homem virou Doutor.

Pra mim, o fim do doutorado significa o encerramento de uma etapa que se iniciou em 1999, na Universidade Federal de Alagoas, quando eu iniciei o curso de graduação em Ciência da Computação. E como é bom o gostinho de encerrar etapas, principalmente quando dá tudo certo.

Mas obviamente que não se constrói nada sozinho, durante o caminho recebi muita ajuda de pessoas que contribuíram não só para a minha formação profissional quanto também para a minha formação pessoal. Foram várias as pessoas que eu deveria agradecer e com certeza esquecerei alguém (afinal de contas, memória nunca foi o meu forte). Por isso, manterei o meu foco de agradecimento no período do Doutorado.

Ao Professor Lucena, pelas várias contribuições técnicas em artigos, as inúmeras revisões na tese e a paciência por ter escutado pelo menos umas 30 apresentações sobre o mesmo tema. Queria agradecer também pelas experiências internacionais que me foram proporcionadas no Canadá, França e Inglaterra. Mas o principal agradecimento ao Lucena é a confiança depositada em mim tanto no aspecto profissional quanto pessoal, o que me permitiu desempenhar várias atividades que contribuíram enormemente para as experiências que vivi aqui no Rio de Janeiro.

Ao Guga ... ihh ... esse aqui merece um agradecimento especial. Foi com o Guga que desenvolvi quase todos os trabalhos na PUC-Rio. Foi com ele que tive várias das discussões de direcionamentos de pesquisa, e até mesmo várias decisões de

cunho profissional. Guga, valeu pela força cumpade!

Ao Professor Ricardo Choren, que contribui para a escrita de vários artigos e também em inúmeras discussões.

A banca examinadora, pelas valiosas contribuições técnicas a este trabalho.

A Vera, por todos os galhos, não, galhos não, árvores inteiras que foram quebradas por esta pessoa incansável.

Aos amigos Galuteta e Hyggo que mesmo distantes colaboraram com esta tese.

A toda a galera do LES que tornou o dia-a-dia uma experiência sempre agradável e produtiva. Em especial, ao Sérgio Ruivace que me ajudou com a implementação do estudo de caso do tráfego aéreo.

A minha família maravilhosa que esteve sempre presente, mesmo geograficamente distante. Um agradecimento super-especial para o Coroa (meu pai), minhas irmãs Kysi e Kely, minha vó Léa, minhas tias Séfora, Kilza, Soraya e Kênea e meu primo Edinho.

E o agradecimento final vai para a minha namorosa (mistura de namorada + esposa) Taíse. Ô ... quanta paciência você teve que ter né? Obrigado por tudo nesse período, obrigado por ficar sempre ao meu lado, tenha certeza que sem você, a estrada seria bem mais esburacada e desagradável.

Ahhh ... e pra cumprir o protocolo, ao CNPq pelo apoio financeiro.

Resumo

Paes, Rodrigo de Barros; Lucena, Carlos José Pereira de. **Governança de Sistemas Multi-Agentes Abertos com Fidedignidade.** Rio de Janeiro, 2007. 197p. Tese de Doutorado - Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Sistemas multi-agentes abertos são freqüentemente caracterizados como sistemas distribuídos onde agentes, que podem não ser conhecidos a priori, podem entrar ou sair do sistema a qualquer momento. Uma abordagem de governança estabelece regras de interação que precisam ser seguidas pelos agentes de um determinado sistema. O uso destas regras permite uma maior previsibilidade e controle sobre o comportamento observável do sistema. Nesta tese, apresenta-se uma abordagem de governança de sistemas multi-agentes abertos como adequada para lidar não apenas com o monitoramento e controle do comportamento dos agentes, mas também com aspectos de fidedignidade. Um software é dito fidedigno quando se pode confiar no mesmo através de verificações formais ou informais assumindo riscos de danos compatíveis com o serviço prestado pelo software. A incorporação de aspectos de fidedignidade em uma abordagem de governança tem como principal benefício a geração de uma tecnologia integrada que possui as vantagens de uma abordagem de governança e ao mesmo tempo lida com especificações de instrumentos de fidedignidade tais como prevenção e tolerância a faltas. A abordagem proposta é ilustrada através de um estudo de caso no contexto de controle de tráfego aéreo.

Palavras-chave

Sistemas Multi-agentes; Sistemas Abertos; Governança baseada em Leis; Leis de Interação; Fidedignidade

Abstract

Paes, Rodrigo de Barros; Lucena, Carlos José Pereira de. **Governance of Open Multi-Agent Systems with Dependability.** Rio de Janeiro, 2007. 197p. Doctoral Thesis - Computer Science Department, Pontifical Catholical University of Rio de Janeiro.

Open multi-agent systems are frequently characterized by having 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. Furthermore, agents may enter or leave the system at their will. A governance approach defines the interaction rules that must be obeyed by the agents. These rules allow for a greater control and predictability of the observable system behavior. In this thesis, we propose a governance approach to deal not only with the monitoring and control of agents' behavior but also to deal with dependability concerns. The original definition of dependability is the ability to deliver service that can justifiably be trusted. A governance approach that also deals with dependability has as main benefit the reuse of the monitoring and enforcement present in the governance infrastructure for dependability. We present a case study in the context of an air traffic control system to illustrate our approach.

Keywords

Multi-Agent Systems; Open Systems; Law-Governed Approach; Dependability.

Sumário

1 Introdução	18
1.1. Problema	21
1.2. Limitações de Abordagens Existentes	21
1.3. Solução Proposta	22
1.4. Contribuições	23
1.5. Organização da Tese	24
 2 Capítulo 6 - Incorporação de Fidedignidade na Abordagem de Governança	24
 3 O Meta-Modelo de Leis: XMLaw	26
3.1. Descrição Detalhada dos Elementos do Modelo Conceitual	29
3.2. Modelo de Eventos	50
3.3. Contextos	52
3.4. Lista de eventos	52
3.5. Gramática	53
3.6. Considerações Finais	55
 4 Trabalhos Relacionados ao Meta-Modelo	56
4.1. Relação do Modelo Conceitual do XMLaw com uma Abordagem de Baixo Nível Baseada em Eventos	59
4.2. Relação do Modelo Conceitual do XMLaw com uma Abordagem de Alto Nível não Baseada em Eventos	66
4.3. Moise +	68
4.4. Estudos de Caso	69
4.5. Considerações Finais	79
 5 Infra-estrutura de Implementação: M-Law	81
5.1. Ciclo de Vida do Mediador MLaw	85
5.2. Arquitetura e Implementação	86

5.3. Trabalhos relacionados ao Middleware	94
5.4. Considerações Finais e Exemplos de Utilização do M-Law	97
6 Fidedignidade	100
6.1. Detecção de Falhas	103
7 Incorporação de Fidedignidade na Abordagem de Governança	105
7.1. Implementação de Duas Estratégias de Tolerância a Faltas Através do XMLaw	105
7.2. Dependability Explicit Computing e Leis	114
8 Estudo de Caso: Controle de Tráfego Aéreo	131
8.1. Leis em CTAs	131
8.2. Fontes de Problemas em Sistemas de CTA	132
8.3. Engenharia do Sistema	134
8.4. Casos de Uso	136
8.5. Arquitetura	142
8.6. Análise da Governança	144
8.7. Análise da Fidedignidade	147
8.8. <i>Dependability Explicit Computing</i>	169
8.9. Resumo dos Artefatos de Lei Gerados no Estudo de Caso	175
8.10. Considerações Finais	181
9 Conclusões	183
9.1. Resultados	184
9.2. Limitações	186
9.3. Trabalhos Futuros	186
Referências	190

Lista de Figuras

Figura 1 – Governança sob a Ótica das Disciplinas de Inteligência Artificial, Ciências Sociais e Engenharia de Software	27
Figura 2 – Modelo Conceitual do XMLaw	28
Figura 3 – Ciclo de Vida de um Elemento de Lei	51
Figura 4 – Exemplo de Protocolo	62
Figura 5 – Exemplo da Necessidade de Sincronização entre os Controladores para a Identificação de Propriedades Globais.	65
Figura 6 – Protocolo de Interação do Estudo de Caso da Equipe de Compradores	73
Figura 7 – Protocolo de Interação do Estudo de Caso Centro de Conferências	77
Figura 8 – Arquitetura do Mediador M-Law	81
Figura 9 – Componentes Principais do M-Law	82
Figura 10 – Projeto do Elemento <i>Scene</i>	84
Figura 11 – API cliente: classes LawFacade e Agent	85
Figura 12 – Ciclo de Vida do Mediador M-Law	86
Figura 13 – Diagrama de Pacotes	87
Figura 14 – Diagrama de Deployment	87
Figura 15 – Diagrama de Classes do Pacote <i>agent</i>	88
Figura 16 – Diagrama de Classes do Pacote <i>communication</i>	90
Figura 17 – Diagrama de Classes do Pacote <i>jade</i>	91
Figura 18 – Diagrama de Classes do Pacote <i>event</i>	92
Figura 19 – Diagrama de classes do Pacote <i>execution</i>	93
Figura 20 – Relacionamento entre falta, erro e falha (Avizienis, Laprie et al. 2004)	107
Figura 21 – Estratégia <i>Forward Recovery</i> (Avizienis, Laprie et al. 2004)	109
Figura 22 – Sistema de Controle de Vendas	110
Figura 23 – Protocolo de Interação do Sistema de Controle de Vendas	111
Figura 24 – Diagrama de Seqüência do Estudo de Caso	118
Figura 25 – Arquitetura do Estudo de Caso de DepEx	119
Figura 26 – Protocolo de Interação do Estudo de Caso de DepEx	124

Figura 27 – Modelo Entidade-Relacionamento do Banco de Dados de Metadados	126
Figura 28 – Exemplos de Agentes Cadastrados no Banco de Dados	127
Figura 29 – Exemplos de Informações de Fidedignidade	127
Figura 30 – Modelagem do Espaço Aéreo em um CTA	134
Figura 31 – Arquitetura do Estudo de Caso	143
Figura 32 – Protocolo de Interação da Cena Controle de Pista	144
Figura 33 - Protocolo de Interação da Cena Decolagem	145
Figura 34 - Protocolo de Interação da Cena Vôo	146
Figura 35 - Protocolo de Interação da Cena Aterrissagem	147
Figura 36 – Arquitetura do Estudo de Caso com a Inclusão do Agente Detector de Falhas	153
Figura 37 – Protocolo de Interação da cena de vôo	158
Figura 38 – Protocolo de Interação da Cena de Vôo	160
Figura 39 – Protocolo de Interação da Cena de Controle de Pista	164
Figura 40 – Protocolo da Cena de Aterrissagem com a inclusão dos estados s5 e s6	167
Figura 41 – Protocolo da Cena de Aterrissagem Modificado com a inclusão dos estados s7 e s8.	168
Figura 42 – GQM do Objetivo G01	172
Figura 43 – Modelo do Banco de Dados	172
Figura 44 – GQM do Estudo de Caso para a Identificação dos Metadados	175
Figura 45 – Arquitetura do Estudo de Caso do Controle de Tráfego Aéreo	176
Figura 46 – Tela da Visão Geral do Sistema	179
Figura 47 – Tela Representando o Agente Piloto	180
Figura 48 – Tela Representando a Visão do Agente Controlador	181

Lista de Tabelas

Tabela 1 – Dimensões de Preocupações de Cada Elemento do XMLaw	29
Tabela 2 – Lista de Eventos	53
Tabela 3 – Lista dos Principais Eventos do LGI	61
Tabela 4 – Lista das Principais Operações do LGI	62
Tabela 5 – Mapeamento entre o XMLaw e o LGI	64
Tabela 6 – Relacionamento entre os Modelos Conceituais de XMLaw e IE	68
Tabela 7 – Linha de Raciocínio para a Especificação XMLaw do Requisito #3 nas Linhas 35, 39, 43 e 45.	122
Tabela 8 – Descrição dos Atributos do Banco de Dados	127
Tabela 9 – Guia de Análise de Governança	144
Tabela 10 – Elicitação das Ameaças	150
Tabela 11 – Estratégias de Mitigação das Ameaças	152
Tabela 12 – Descrição dos Atributos do Banco de Dados	173
Tabela 13- Exemplo de Tupla Inserida no Banco de Dados	173

Lista de Código

Código 1 – Exemplo de Clock Ativado e Desativado por Transições	31
Código 2 – Exemplo de Clock Ativado por uma Transição ou Norma.	32
Código 3 – Exemplo do Papel Utilizado em uma Mensagem para Definir o Remetente e o Destinatário	32
Código 4 – Exemplo do Papel Utilizado para Definir os Criadores e Participantes de uma Cena	33
Código 5 – Exemplo da Especificação do Agente em uma Mensagem XMLLaw	34
Código 6 – Exemplo de Mensagem Simples	35
Código 7 – Exemplo de Mensagem com Representação dos Agentes	35
Código 8 – Exemplo de Mensagem com a Performativa Explícita	36
Código 9 – Exemplo de Mensagem Complexa	36
Código 10 – Exemplo do Elemento Law	37
Código 11 – Exemplo de uma Cena com Criadores, Participantes, Mensagens, Estados e Transições.	39
Código 12 – Exemplo de Norma	41
Código 13 – Exemplo de uma <i>Constraint</i> Utilizada em uma Transição	43
Código 14 – Código Java de uma Constraint	43
Código 15 – Exemplo de Protocolo com o Nome Declarado	45
Código 16 – Exemplo de Protocolo sem a Declaração do Nome	45
Código 17 – Exemplo de Lei que Declara uma Action	49
Código 18 – Exemplo de Implementação de Action	49
Código 19 – Pseudocódigo para a Ativação de uma Norma Devido ao Disparo de uma Transição	50
Código 20 – Pseudocódigo para Compor uma Norma com um <i>Clock</i> .	51
Código 21 – Definição dos Eventos que Ativam um Elemento de Lei	52
Código 22 – Gramática XMLLaw	55
Código 23 – Mapeamento do Protocolo da Figura 4 usando XMLLaw	63

Código 24 – Mapeamento do Protocolo da Figura 4 usando LGI	63
Código 25 – Redirecionamento de Mensagens para um Mediador Central	66
Código 26 – XMLaw do Estudo de Caso da Equipe de Compradores	73
Código 27 – <i>Constraint</i> que Verifica se o Agente é de Fato o Gerente Atual	74
Código 28 – <i>Action</i> que Troca o Gerente Atual	74
Código 29 – <i>Constraint</i> que Verifica se o Agente que está Cedendo Dinheiro Realmente tem Dinheiro para Ceder	74
Código 30 – <i>Action</i> que Atualiza os Orçamentos e que é Utilizado nas Mensagens de <i>giveBudget</i> e <i>removeBudget</i>	75
Código 31 –XMLaw da Cena <i>appointmentProposal</i>	78
Código 32 – Pseudocódigo do Sistema Distribuído Ping-Pong	102
Código 33 – Processo Pong com a Inclusão de uma Falta	103
Código 34 – Especificação da Lei do Sistema de Controle de Vendas	111
Código 35 – XMLaw do Estudo de Caso de DepEx	125
Código 36 – Implementação Java da <i>constraint checkContent</i>	126
Código 37 – <i>Action updateClockMetadata</i> implementada através da classe Java <i>DecAvailability</i>	126
Código 38 – Protocolo de interação em XMLaw da cena <i>groundControl</i>	145
Código 39 – Protocolo de interação em XMLaw da cena <i>take-off</i>	145
Código 40 – Protocolo de Interação em XMLaw da cena <i>flight</i>	147
Código 41 – XMLaw da cena de vôo com a estratégia de mitigação da ameaça CDL01	154
Código 42 – Implementação da Constraint <i>CheckDistance</i>	155
Código 43 – XMLaw da Cena de Vôo com a Estratégia de Mitigação da Ameaça CDL02	156
Código 44 – XMLaw da Cena de Decolagem com a Estratégia de Mitigação da Ameaça CDL03	157
Código 45 – XMLaw da Cena de Vôo com a Estratégia de Mitigação da Ameaça CDL05	159
Código 46 – XMLaw da Cena de Vôo com a Estratégia de Mitigação da Ameaça CDL06	161
Código 47 – XMLaw Utilizando o Agente Detector de Falhas	162
Código 48 – Utilização do Agente Detector de Falhas para Reportar Falhas de	

Comunicação com os Radares.	163
Código 49 – XMLaw de Mitigação da Ameaça CDL09.	164
Código 50 – XMLaw de Mitigação da Ameaça CDL11	166
Código 51 – XMLaw de Mitigação da Ameaça CDL12	167
Código 52 – XMLaw de Mitigação da Ameaça CDL13	169
Código 53 – Cena <i>flight</i> Modificada com a Inserção da <i>action</i> para Armazenar os Dados no Banco de Dados	174

Lista de Siglas e Abreviaturas

IE *Instituição Eletrônica*

UML *Unified Modeling Language*

SMA *Sistema Multi-Agente*

CTA *Controle de Tráfego Aéreo*

1

Introdução

A tecnologia de agentes de software é considerada como adequada para lidar com a heterogeneidade, adaptabilidade e a característica aberta de alguns sistemas distribuídos (Hannoun, Boissier et al. 2000; Dignum and Dignum 2001; DeLoach and Matson 2004). 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. Pode-se dizer que sistemas abertos são concorrentes, uma vez que são compostos por diferentes subsistemas operando geograficamente distribuídos e utilizam-se de diferentes fontes de informações; assíncronos; possuem controle descentralizado, no sentido de que o fluxo de execução do sistema é regido por decisões locais dos subsistemas; e podem conter informação inconsistente (Hewitt 1986). 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 (Guttman, Moukas et al. 1998; Ripper, Fontoura et al. 2000), cadeia de fornecimento (Michael, Alberto et al. 2006) e aplicações médicas (Lapinsky, Weshler et al. 2004).

Muitos pesquisadores utilizam 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, consequentemente,

diminuir a probabilidade de falhas no sistema. Neste contexto, vários modelos de instituições eletrônicas foram propostos (Hannoun, Boissier et al. 2000; Minsky and Ungureanu 2000; Dignum 2002; Esteva 2003; Esteva, Rosell et al. 2004).

O termo organização de agentes também é bastante utilizado na literatura. Uma organização de agentes pode ser definida como um padrão de cooperação emergente ou pré-definida entre os agentes de um sistema. Este padrão pode ser definido pelo projetista do software ou pelos próprios agentes (Boissier, Hübner et al. 2006).

Neste trabalho, utiliza-se o termo governança para reunir 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 e alguns tipos de organizações são consideradas como uma abordagem de governança.

Por outro lado, artigos recentes enfatizam que software continua sofrendo pela falta de qualidade (Kan 2002; Noushin 2003; Mahmood, David et al. 2005). Sabe-se que cerca de 50% do software tornado disponível contém falhas não triviais (Boehm and Basili 2001). Ainda assim, 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 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 faltas¹ e assegurar corretude, entre outras propriedades. O entendimento da noção de fidedignidade² depende do serviço que o software oferece. Em alguns serviços como o comando e controle, uma falha pode ter consequências 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 freqüência 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 confiar no mesmo através de verificações formais ou informais assumindo riscos de danos compatíveis com o serviço prestado pelo software (Avizienis, Laprie et al. 2004).

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 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 as vantagens de uma abordagem de governança e ao mesmo tempo lida com especificações de instrumentos para alcançar maiores níveis de fidedignidade.

Desta forma, a principal contribuição deste trabalho situa-se 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 agragar e expressar os conceitos relacionados a governança encontrados na literatura atual; (iii) possuir um alto nível de abstração de seus elementos e ainda

¹ A presença de faltas pode levar a ocorrência de falhas no sistema. Uma definição mais precisa sobre estes termos será apresentada no capítulo 5.

² A palavra fidedignidade foi a tradução encontrada pelo Prof. Arndt Von Staa Staa, A. v. (2006). Engenharia de Software Fidedigno. Monografias em Ciência da Computação, n. 13/06. PUC-Rio. Rio de Janeiro, Pontifícia Universidade Católica do Rio de Janeiro. para o termo inglês *dependability*.

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

O principal problema abordado neste trabalho é a falta de garantia 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, existe uma carência para o estabelecimento de 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) os 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 (Minsky 2005); (ii) ou são de um nível de abstração tão alto que não permitem o mapeamento automático da especificação para um mecanismo de implementação (Dignum 2002); (iii) ou possuem um conjunto de elementos adequados, porém baseado em modelos conceituais que tornam fixa a forma de composição entre os seus elementos e não prevêem mecanismos para evoluir o próprio modelo conceitual, o que dificulta a adaptação da abordagem para evoluções tanto das leis quanto da abordagem em si (Esteva 2003; Hübner 2003). É importante destacar que a limitação relacionada ao item

(iii) não se dá pela inflexibilidade ou pela falta de expressividade dos elementos que compõem seus modelos conceituais, mas sim pela própria estrutura de composição acoplada entre esses elementos. Suponha, por exemplo, que se deseje incluir um elemento de tratamento de exceções em um destes modelos conceituais; isso não seria possível sem uma revisão estrutural em todos os elementos que seriam afetados pela entrada deste novo conceito. Finalmente, (iv) não foram encontradas abordagens que incorporem preocupações de fidedignidade.

1.3. Solução Proposta

Dante 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 (Paes, Carvalho et al. 2007). Outros elementos foram introduzidos conforme a necessidade dos experimentos realizados e relatados em (Carvalho, Paes et al. 2005; Carvalho, Paes et al. 2005; Paes 2005; Rodrigues, Carvalho et al. 2005; Carvalho, Almeida et al. 2006; Gatti, Carvalho et al. 2006; Gatti, Lucena et al. 2006; Gatti, Paes et al. 2006). 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 do próprio modelo conceitual, 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 em 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 trabalho³:

- (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 (Paes 2005; Paes, Carvalho et al. 2005) e utiliza-se de analogias para os conceitos de programação por convenção (do inglês *coding by convention*) de projetos como Rails (Thomas, Hansson et al. 2006) e Grails (Rocher 2006) 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. A incorporação de preocupações de fidedignidade em uma abordagem de governança é certamente a principal contribuição deste trabalho;
- (vi) Um estudo de caso realista que ilustra que a abordagem de leis pode ser aplicada para construir sistemas complexos mais confiáveis.

³ Os itens de (i) a (iv) são referidos como a abordagem proposta.

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 usados para especificar uma lei. É apresentada a semântica, o relacionamento e a forma de representação em uma linguagem declarativa de cada um dos elementos.
Capítulo 3 - Trabalhos Relacionados ao Meta-Modelo	Neste capítulo, apresentam-se as características 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 - Fidedignidade	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 - Incorporação de Fidedignidade na Abordagem de Governança	Neste capítulo, apresenta-se como a fidedignidade pode ser incorporada na abordagem de governança proposta. Mostra-se como implementar as estratégias <i>Full Forward Recovery</i> e <i>Partial Forward Recovery</i> . Apresenta-se também o conceito de <i>dependability explicitly computing</i> e uma implementação para ilustrar como realizar <i>dependability explicit</i>

	<i>computing</i> utilizando a abordagem proposta
Capítulo 7 - Estudo de Caso: Controle de Tráfego Aéreo	Neste capítulo, apresenta-se um estudo de caso que ilustra a aplicação de leis. Discute-se também, como as leis estão ajudando a alcançar um maior grau de fidedignidade para este exemplo.
Capítulo 8 - Conclusões	Neste capítulo, são apresentadas reflexões sobre o trabalho realizado e discussões sobre pontos em aberto e trabalhos futuros.

2 O Meta-Modelo de Leis: XMLaw

O problema de estabelecer governança em sistemas multi-agentes é abordado sob diferentes pontos de vistas por comunidades tais como Ciências Sociais, Inteligência Artificial e Engenharia de Software. Os cientistas sociais estão predominantemente interessados em utilizar agentes para modelar o comportamento humano individual e coletivo. A principal idéia é desenvolver modelos organizacionais e representá-los como modelos computacionais para a realização de experimentos. O resultado obtido com os experimentos é usado para validar os modelos organizacionais. Os cientistas sociais utilizam governança como um elemento chave para a inteligência coletiva, ou seja, ela representa o conhecimento social e cultural através de normas, protocolos e leis. A Inteligência Artificial preocupa-se predominantemente com a reutilização destes modelos sociais para a construção de sistemas inteligentes. A pesquisa nesta área constrói modelos computacionais para representar modelos sociais e desenvolve um conjunto de teorias, modelos de inferência, planos e até mesmo emoções artificiais para construir sistemas mais adaptativos e preparados para evolução. Neste contexto, o papel principal da Engenharia de Software é sistematizar a construção destes sistemas inteligentes de tal forma que os resultados levarão a sistemas que não são apenas mais inteligentes, mas principalmente fáceis de manter, evoluir e mais confiáveis.

Claramente existe uma complementaridade entre as três áreas citadas. O papel da Engenharia de Software é crucial para que a tecnologia de governança possa se tornar madura através de metodologias e ferramentas para a construção de sistemas de qualidade (Figura 1). O trabalho apresentado nesta tese se atém predominantemente na pesquisa de governança sob a ótica de Engenharia de Software.

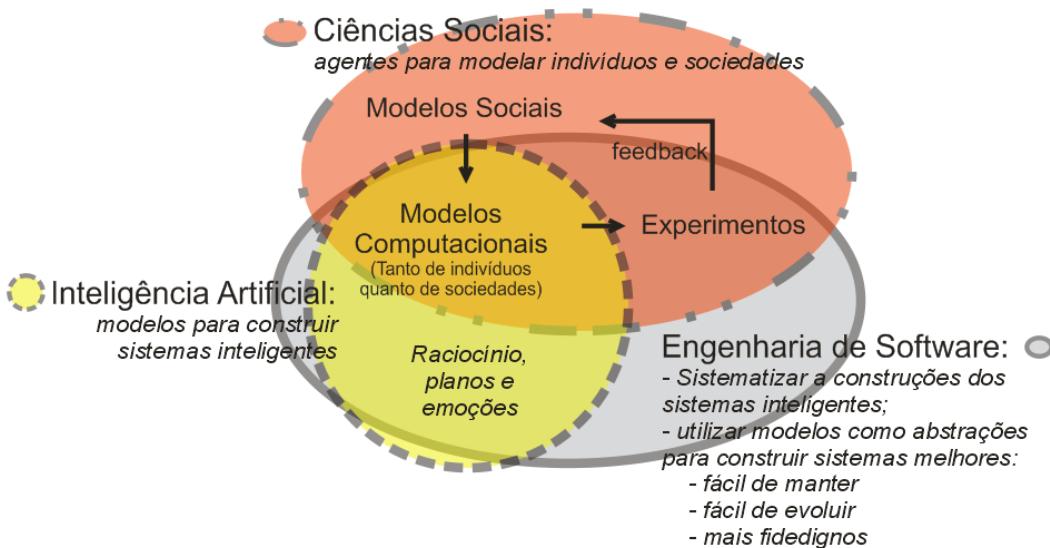


Figura 1 – Governança sob a Ótica das Disciplinas de Inteligência Artificial, Ciências Sociais e Engenharia de Software

Neste capítulo, apresenta-se o modelo conceitual, o modelo de eventos e a linguagem de especificação do XMLaw. Na **Figura 2**, mostram-se os elementos que compõem o modelo conceitual do XMLaw (Paes 2005; Paes, Carvalho et al. 2005). O termo modelo conceitual possui o mesmo significado do que a OMG⁴ normalmente utiliza para referenciar o meta-modelo de UML. Este modelo é composto de elementos que buscam englobar as várias dimensões de uma especificação de leis. A Tabela 2 mostra os elementos do modelo conceitual categorizados de acordo com uma dimensão de preocupação, seguindo a taxonomia proposta em (Coutinho, Sichman et al. 2005).

⁴ OMG – Object Management Group – www.omg.org

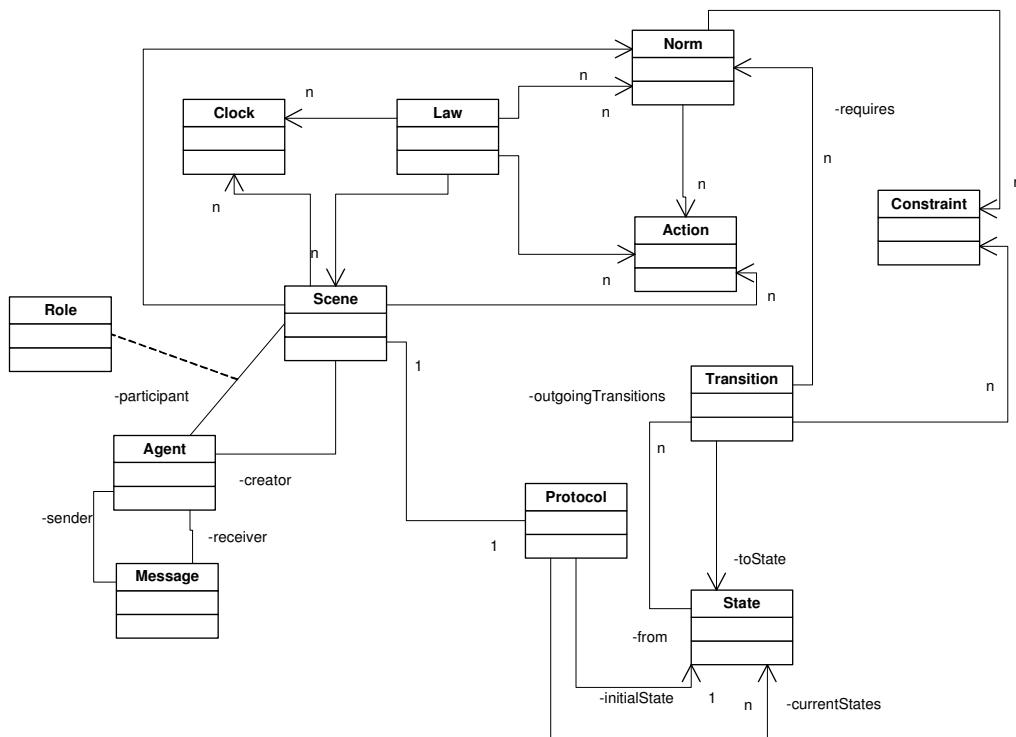


Figura 2 – Modelo Conceitual do XMLaw

	Estrutural	Interação	Funcional	Normativa
Clock	X		X	
Role	X			
Agent	X			
Message	X	X		
Law	X			
Scene		X		
Norm				X
Constraint				X
Protocol		X		
State		X		
Transition		X		
Action			X	

Tabela 1 – Classificação dos elementos do modelo conceitual segundo a taxonomia proposta em (Coutinho, Sichman et al. 2005).

Sob uma ótica diferente de análise, os elementos de leis poderiam ser categorizados de acordo com a seguinte classificação:

Temporal – esta dimensão permite a especificação de leis que são sensíveis ao tempo. Por exemplo, algumas regras podem conter prazos ou até mesmo comportamento cíclicos que dependem do tempo.

Social – esta dimensão prevê a existência de relacionamentos sociais e interativos entre os agentes. Exemplos de relacionamento sociais são mestre-escravo e empregador-empregado.

Estrutural – esta dimensão engloba todos os tipos de estrutura utilizada para descrever as leis. Estas estruturas geralmente definem contextos modulares que definem o escopo de validade de partes da lei.

Restritivo – esta dimensão contém elementos que focam em um conjunto restrito de ações que os agentes podem realizar em um determinado contexto.

Serviço – esta dimensão está relacionada com a interação entre as leis e os serviços que existem em um ambiente.

A Tabela 2 mostra os elementos do modelo conceitual categorizados de acordo com esta nova dimensão de preocupação.

Temporal	Social	Estrutural	Restritivo	Serviço
Clock	Role Agent Message	Law Scene	Norm Constraint Protocol State Transition	Action

Tabela 2 – Dimensões de Preocupações de Cada Elemento do XMLaw

2.1. Descrição Detalhada dos Elementos do Modelo Conceitual

O objetivo desta seção é descrever os elementos que compõem o modelo conceitual do XMLaw. A estrutura de apresentação dos elementos é baseada na forma utilizada para especificar a UML (Group 2007). O modelo conceitual do XMLaw foi estendido algumas vezes para contemplar características bem específicas de alguns domínios, tais como análise de criticalidade (Gatti, Lucena

et al. 2006) e testes (Rodrigues, Carvalho et al. 2005). Estas extensões estão fora do escopo deste documento.

2.1.1. Convenções Utilizadas

Enumerações:

Sintaxe: {elemento1, elemento2, ..., elementoN}

Semântica: Uma enumeração que pode assumir qualquer valor definido entre as chaves.

Multiplicidade:

[1]: Exatamente 1;

[0..1]: zero ou um;

[1..*]: pelo menos 1;

[0..*]: qualquer número de elementos.

Tipos de dados:

long: qualquer número inteiro compreendido entre $-2^{64}/2 - 1$ e $+2^{64}/2 - 1$;

String: qualquer cadeia de caracteres;

Boolean: comprehende os valores *true* e *false*

Id: representa um identificador e o seu valor pode ser qualquer cadeia de caracteres. Apesar de estruturalmente um Id ser igual a uma String, optou-se por criar tipos diferenciados para realçar a semântica de identificador presente no tipo Id.

2.1.2. Clock

O elemento *clock* permite representar restrições de tempo em uma interação. O *clock* pode ser utilizado como controle para ativar ou desativar outros elementos. Uma vez ativo, o *clock* pode gerar eventos de *clock_tick*.

ATRIBUTOS

type: {periodic, regular}

Existem dois tipos de *clock*. O *clock* declarado como “*regular*” gera um único evento após o intervalo de tempo especificado. O *clock* do tipo “*periodic*” gera cicличamente eventos, onde cada ciclo é definido pelo intervalo de tempo especificado.

tick-period: long

Especifica um período de tempo em milisegundos. Este tempo é utilizado pelo *clock* para gerar eventos do tipo *clock_tick*.

ASSOCIAÇÕES COM OUTROS ELEMENTOS DO XMLAW

activations: Event[0..]*

Lista de eventos que ativam o *clock*

deactivations: Event[0..]*

Lista de eventos que desativam o *clock*

EVENTOS GERADOS

clock_activation

Evento gerado quando ocorre a ativação de um *clock*. A partir deste momento, o *clock* começa a contar o tempo.

clock_timeout

Ocorre quando um *clock* do tipo “regular” gera um evento do tipo *clock_tick* e, portanto, terminou a sua execução.

clock_tick

Ocorre quando o período de tempo especificado pelo atributo *tick-period* passou.

clock_deactivation

Ocorre quando qualquer uma das condições de desativação do *clock* ocorre e, portanto, a execução do *clock* é interrompida.

EXEMPLO

O *clock* do Código 1 é um *clock* cujo atributo *tick-period* possui o valor de 5000 milisegundos, ele é do tipo regular, e é ativado pelos eventos de ativação de transição (*transition_activation*) gerados pelas transições t1 ou t2 e é desativado por um evento *transition_activation* gerado pela transição t3.

```
clock1{5000,regular, (t1,t2),(t3)}
```

Código 1 – Exemplo de Clock Ativado e Desativado por Transições

É importante perceber que no caso de elementos que geram apenas um tipo de evento, como é o caso da transição, não é preciso declarar o tipo de evento na ativação ou desativação do *clock*. Mas, por exemplo, se um *clock* fosse ativado por uma norma é necessário informar qual o evento que ativará ou desativará o *clock*. No Código 2, mostra-se um exemplo de *clock* do tipo *regular*, que é ativado pelo evento de *transition_activation* gerado pela transição t1 ou por um evento de *norm_activation* gerado pela norma *norm1*, e é desativado por um evento de *transition_activation* gerado pela transição t3.

```
clock2{5000,regular, (t1, (norm1,norm_activation)),(t3)}
```

Código 2 – Exemplo de Clock Ativado por uma Transição ou Norma.

2.1.3.Role

Geralmente as leis são especificadas utilizando-se os papéis (*role*) ao invés dos agentes individuais que desempenham estes papéis. Um papel é uma representação das responsabilidades, habilidades e comportamento esperado de um determinado agente ou grupo de agentes. Esta abstração é bastante útil para omitir detalhes individuais dos agentes que estão desempenhando o papel.

Os papéis são utilizados para especificar o destinatário ou o remetente da mensagem, para definir quem pode participar de uma determinada cena e para definir quem pode criar uma determinada cena.

EXEMPLO

O Código 3 mostra o papel sendo utilizado em uma mensagem. Esta mensagem é uma mensagem enviada por um agente com o papel de *master* cujo destinatário é um outro agente com o papel de *slave*.

```
message01{master, slave, performative(content)}
```

Código 3 – Exemplo do Papel Utilizado em uma Mensagem para Definir o Remetente e o Destinatário

O Código 4 mostra uma cena em que somente agentes desempenhando o papel *buyer* podem criar a cena, agentes desempenhando o papel de *seller* podem entrar na cena somente se o protocolo da cena estiver no estado s0 ou s1 e o *buyer* só pode entrar se o protocolo estiver no estado s1.

```

scene01{
    creator{buyer}
    participant{seller, (s0,s1)}
    participant{buyer, (s1)}
    ...
}

```

Código 4 – Exemplo do Papel Utilizado para Definir os Criadores e Participantes de uma Cena

2.1.4.Agent

Este elemento representa um agente de software que interage com outros agentes sob as regras definidas na lei. Não é feita nenhuma suposição sobre a arquitetura interna ou sobre a linguagem que os agentes são implementados.

O elemento *agent* pode ser utilizado para representar um agente que se conhece a priori. Por exemplo, pode-se especificar uma lei em que um dos agentes é o agente “Banco dos Agentes” e escrever leis tal como: “O *Banco dos Agentes* pode comprar produtos que custem mais de um milhão de reais”.

ATRIBUTOS

identification: String

Identificação única do agente. Geralmente as leis são especificadas independentemente dos agentes, mas considerando-se apenas os papéis. Por exemplo, uma norma pode descrever que “um comprador é obrigado a pagar pelo produto dentro de um período de no máximo dois dias”. Esta norma não possui nenhuma referência a um agente em particular, ao invés disso ela é geral para todos os agentes que desempenham o papel de comprador.

ASSOCIAÇÕES COM OUTROS ELEMENTOS DO XMLAW

message: Message[0..]*

O conjunto de mensagens enviadas pelo agente. Esta associação não é especificada a priori porque não é possível saber a priori quais as mensagens que um agente irá enviar. Entretanto, esta associação está presente no

modelo conceitual para deixar claro o relacionamento entre agentes e mensagens.

roles: Role[0..]*

Um agente pode desempenhar muitos papéis durante o curso da interação. É preciso identificar sob quais papéis o agente irá interagir em uma cena para que as normas e outros elementos de leis que se referem aos papéis possam ser interpretados corretamente.

EXEMPLO

Em casos especiais pode-se determinar o agente específico que enviará uma dada mensagem. O Código 5 mostra uma mensagem enviada pelo agente identificado pelo endereço: *agent://192.82.24.111:7574/bill*.

```
message01{ (master, agent://192.82.24.111:7574/bill), slave,
performative(content) }
```

Código 5 – Exemplo da Especificação do Agente em uma Mensagem XMLaw

2.1.5.Message

Este elemento modela uma mensagem trocada entre agentes. Uma mensagem é definida como a tupla:

$$M=\{U,I,S,SR,R,C,P\}$$

Onde,

U: é a performativa utilizada na mensagem;

I: é uma identificação da conversação;

S: é a identificação do remetente da mensagem;

SR: é o papel desempenhado pelo remetente da mensagem no momento do envio;

R: é o conjunto de destinatários identificados pelos pares de valores {Destinatário, Papel do destinatário};

C: o conteúdo da mensagem;

P: o nome do protocolo de interação utilizado pela mensagem, se existir.

ATRIBUTOS

performative: String

A performativa.

content: String

O conteúdo da mensagem.

sender: AgentRole

O remetente da mensagem.

receivers: AgentRole[1..]*

Os destinatários

EVENTOS GERADOS

message_arival

Evento gerado quando uma mensagem enviada por um agente chega ao mediador

compliant_message

Evento gerado quando a mensagem possui o padrão de mensagem esperado.

EXEMPLO

O Código 6 mostra um exemplo de mensagem simples, contendo apenas o papel do remetente, o papel do destinatário e o conteúdo.

```
message01{seller, buyer, inform(price=500)}
```

Código 6 – Exemplo de Mensagem Simples

O Código 7 mostra um exemplo de mensagem contendo também os endereços dos agentes.

```
message02{(seller, agent://192.82.24.111:7574/bill), (buyer,
agent://192.82.24.111:7574/John), inform(price=500)}
```

Código 7 – Exemplo de Mensagem com Representação dos Agentes

O Código 8 mostra um exemplo de mensagem contendo a definição explícita da performativa

```
message02{(seller, agent://192.82.24.111:7574/bill), (buyer,
agent://192.82.24.111:7574/John), inform, price=500}
```

Código 8 – Exemplo de Mensagem com a Performativa Explícita

O Código 9 mostra um exemplo do formato de mensagem mais completo possível, ela contém os papéis e endereços dos agentes remetentes e destinatários, mais de um destinatário, a performativa, um identificador da conversação e o nome do protocolo de interação

```
message02{
    (seller, agent://192.82.24.111:7574/bill),
    (
        (buyer, agent://192.82.24.111:7574/John),
        (buyer, agent://192.82.24.111:7574/John)
    ),
    inform,
    price=500,
    convId=12,
    protocol=contract-net
}
```

Código 9 – Exemplo de Mensagem Complexa

2.1.6.Law

Este elemento representa o contexto mais externo onde todos os outros elementos são agrupados. Este é o elemento mais geral e não está contido em nenhum outro elemento.

O elemento *Law* encapsula um conjunto de regras, chamadas de leis. Uma lei é composta de *actions*, *norms*, *clocks* e *scenes*. Uma *scene* por sua vez, também pode ser composta de *actions*, *norms* e *clocks*. Desta forma, o elemento *Law* pode ser utilizado para agrupar um conjunto de cenas relacionadas e utilizar as *actions*, *norms* e *clocks* para capturar o comportamento que estão além do escopo de uma única cena. Por exemplo, se um agente em uma cena desobedecer uma regra importante, ele poderia ser punido com uma proibição de interagir em qualquer cena no escopo da lei. Esta proibição pode ser especificada no contexto do elemento *Law* e, portanto, estaria acessível a todas as cenas (*scenes*).

ATRIBUTOS

name: String

O nome da lei

ASSOCIAÇÕES COM OUTROS ELEMENTOS DO XMLAW

*scenes: Scene[0.. *]*

Todas as cenas que fazem parte da lei

*actions: Action[0.. *]*

Todas as *actions* que podem ser executadas no contexto da lei. Estas *actions* são capazes de perceber os eventos que ocorrem tanto no escopo da lei quanto no escopo de cada uma das cenas que compõem a lei.

*norms: Norm[0.. *]*

Normas que existem no contexto da lei.

*clocks: Clock[0.. *]*

Clocks declarados no contexto da lei.

EXEMPLO

O Código 10 mostra o exemplo de como este elemento é especificado.

```
murphLaw{
    scene01{
        ...
    }

    scene02{
        ...
    }

    action01{...}
    norm01{...}
    clock01{...}
}
```

Código 10 – Exemplo do Elemento Law

2.1.7.Scene

O modelo conceitual utiliza a abstração de cenas para auxiliar na organização e modularização das interações. A idéia de cenas é análoga a uma cena de peça de teatro. Em peças de teatro, os atores atuam de acordo com roteiros (*scripts*) bem definidos e a peça é composta de várias cenas conectadas seqüencialmente. Esta analogia foi apresentada (Esteva 2003) e é utilizada nesse trabalho com algumas modificações. Cenas são representadas pelo elemento do modelo conceitual *Scene*. Este elemento especifica quais agentes e quais os papéis de agentes podem interagir em uma cena, ou mesmo dar início a sua execução.

Além disso, uma cena é composta por um protocolo de interação e por um conjunto de normas, *actions* e relógios (*Clock*). Estes elementos compartilham um contexto comum de interação definido pela cena. Isto significa que uma norma definida no contexto de uma cena é somente visível naquela cena.

ATRIBUTOS

time-to-live: long

Tempo especificado em milisegundos que significa o tempo máximo que uma cena deve durar. Se a cena ainda estiver em execução quando o tempo expirar, gera-se um evento do tipo *time_to_live_expired*.

Embora este atributo seja do tipo *long*, é possível especificar os valores utilizando os seguintes caracteres:

- d: representa 1 dia (86400000 milisegundos)
- w: representa 1 semana (604800000 milisegundos)
- *infinity*: significa que a cena nunca expira.

ASSOCIAÇÕES COM OUTROS ELEMENTOS DO XMLAW

protocol: Protocol[1]

O protocolo de interação desta cena.

actions: Action[0..]*

As *actions* que podem ser executadas no contexto da cena. Estas *actions* só podem escutar eventos que são gerados no contexto da cena.

norms: Norm[0..]*

Normas que existem no contexto da cena.

clocks: Clock[0..]*

Clocks declarados no contexto da cena.

participants: Participant[0..]*

Lista de todos os participantes que podem entrar na cena. O elemento *Participant* não faz parte do modelo conceitual. Ele é utilizado como um auxílio para tornar mais clara a explicação do modelo. Sua estrutura é descrita na Seção 2.1.14.

creators: Creator[0..]*

Especifica quais agentes podem criar uma instância da cena. As interações sempre ocorrem no escopo de uma instância de cena. Várias instâncias da mesma cena podem estar em execução ao mesmo tempo. A estrutura do elemento auxiliar *Creator* é descrita na Seção 2.1.14.

EVENTOS GERADOS

scene_creation

Evento gerado quando a cena é criada.

time_to_live_elapsed

Evento gerado após terem se passado t+1 milisegundos do atributo *time-to-live*.

failure_scene_completion

Representa que a cena terminou através de um estado final do protocolo cujo tipo é de falha.

successful_scene_completion

Representa que a cena terminou através de um estado final do protocolo cujo tipo é de sucesso.

EXEMPLO

O Código 11 mostra um exemplo de uma cena contendo vários dos seus atributos e associações.

```
scene01{
    creator{buyer}
    participant{seller, (s0,s1)}
    participant{buyer, (s1)}

    ping-message{ping, pong, inform(ping) }
    pong-message{pong, ping, inform(ping-pong) }

    s0{initial}
    s2{success}

    t1{s0->s1, ping-message}
    t2{s1->s2, pong-message}
}
```

Código 11 – Exemplo de uma Cena com Criadores, Participantes, Mensagens, Estados e Transições.

2.1.8.Norm

Existem três tipos de normas no XMLaw: obrigações, permissões e proibições. Uma proibição geralmente representa um compromisso que os agentes de software adquiriram durante a interação com outras entidades. Por exemplo, o vencedor de um leilão é obrigado a pagar pelo bem comprado. Esta obrigação vem acompanhada de penalidades que evitam com que ela seja descumprida. A permissão define os direitos de um agente em um determinado momento, por exemplo, o vencedor de um leilão tem permissão para interagir com o banco através do protocolo de pagamento. Finalmente, uma proibição define as ações que não são permitidas, por exemplo, se um agente não quitar seus débitos, ele não poderá participar de interações em nenhuma cena.

ATRIBUTOS

type: {permission, obligation, prohibition}

O tipo da norma

ASSOCIAÇÕES COM OUTROS ELEMENTOS DO XMLAW

assignee: Assignee[1]

O agente que receberá a norma. A estrutura do elemento auxiliar *Assignee* é descrita na Seção 2.1.14.

activations: Event[0..]*

Lista de eventos que ativam a norma.

deactivations: Event[0..]*

Lista de eventos que desativam a norma

constraints: Constraint[0..]*

O relacionamento entre normas e *constraints* foi introduzido em (Carvalho 2007) e significa que enquanto as *constraints* retornarem *false* a norma permanece válida. Se qualquer uma das *constraints* retornar *true* então a norma não é mais considerada válida.

actions: Action[0..]*

Este relacionamento também foi introduzido em (Carvalho 2007). Trata-se do conjunto de *actions* que serão executadas, independentemente uma das outras, enquanto a norma estiver ativa e os eventos de ativação das *actions* ocorrerem.

EVENTOS GERADOS

norm_activation

Evento gerado quando ocorre a ativação da norma.

norm_deactivation

Evento gerado quando ocorre a desativação da norma.

EXEMPLO

O exemplo mostrado no Código 12 mostra uma norma que é ativada pelo evento *transition_activation* gerado pela transição t3 e desativada pelo evento *transition_activation* gerado pela transição t4.

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

Código 12 – Exemplo de Norma

2.1.9.Constraint

Constraints são restrições utilizadas em normas ou transições. Elas especificam filtros, restringindo o conjunto de valores permitidos para um determinado atributo de um evento. As mensagens possuem informações que são verificadas de várias formas: a declaração da mensagem no XMLaw especifica o padrão esperado das mensagens, entretanto, este padrão não detalha quais os valores específicos que cada uma das eventuais variáveis pode conter. Por exemplo, o padrão para o atributo *content* de uma mensagem poderia ser expresso como: *content(television,brand(\$anyBrand),price(\$amount))*. Porém, uma determinada aplicação pode requerer que o valor da variável *\$amount* possua um valor entre 50 e 100. Somente através da especificação do padrão da mensagem

este requisito não pode ser cumprido. O elemento *Constraint* pode ser utilizado para este propósito.

As *constraints* são implementadas utilizando-se código Java. Define-se através do atributo *class* qual a classe Java que irá implementar o filtro. A classe Java deverá estar acessível ao mediador. A classe será chamada pelo mediador quando uma transição ou norma que referencia a *constraint* estiver em condições de executar. A implementação de uma *constraint* precisa implementar o método *constrain(ReadonlyContext ctx):boolean*. Se este método retornar *true*, significa que o filtro foi aplicado e, no caso onde a *constraint* é utilizada por uma transição, a transição não é disparada. Este método recebe como parâmetro o contexto no qual a *constraint* está inserida. Através deste contexto, é possível recuperar as variáveis tais como quem foi o agente remetente da mensagem que ativou a *constraint*, dentre outras.

ATRIBUTOS

class: String

O nome da classe Java que implementa a lógica da *constraint*.

EVENTOS GERADOS

constraint_not_satisfied

Evento gerado quando ocorre a ativação de uma *constraint*, ou seja, o seu método *constrain* retorna *true*.

EXEMPLO

A linha 20 do Código 13 ilustra a declaração de uma *constraint*. Esta *constraint* é utilizada na linha 16. Ou seja, a transição t9 só dispara se a *constraint* *checkContent* retornar *false*. O código Java de uma *constraint* é mostrado no Código 14.

```
16:   t9{s7->s3, msg1, [checkContent]}
17:   t10{s7->s8, timeout2}
...
// Constraints
20: checkContent{br.pucrio.CheckContent}
```

Código 13 – Exemplo de uma *Constraint* Utilizada em uma Transição

```
class CheckContent 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;
    }
}
```

Código 14 – Código Java de uma Constraint

2.1.10.Protocol

O protocolo é um autômato finito não-determinístico (P) definido pela tupla:

$$M = (E, S, T, s_0, F)$$

Onde:

E: um conjunto finito de símbolos de entrada, cujo membros do conjunto são os eventos do XMLaw (ver na Seção 2.4 a lista completa de eventos);

S: um conjunto finito de estados;

T: uma função de transição definida como:

$$T: E \times S \rightarrow 2^S$$

s_0 : o estado s_0 , definido como estado inicial tal que s_0 pertence a S;

F: um conjunto de estados finais tal que F é subconjunto de S.

Um protocolo define os possíveis estados pelos quais a interação pode evoluir. Transições entre estados podem ser disparadas por qualquer evento do XMLaw, ao invés de somente a chegada de mensagens (*message_arrival*). Um autômato finito não-determinístico (Menezes 1997) pára a sua execução após consumir todos os símbolos do alfabeto de entrada e fornece duas saídas:

- ACEITAR, se ao final do processamento existe pelo menos um estado final na lista de estados possíveis; ou
- REJEITAR, quando ao final do processamento não existe nenhum estado final na lista de possíveis estados.

Neste sentido, um autômato finito não pára quando se alcança um estado final, mas sim quando não existem mais símbolos no alfabeto de entrada para ler. Ao utilizar um autômato para representar as interações, pode-se considerar que as mensagens são os símbolos do alfabeto de entrada. Em um sistema multi-agentes aberto, os agentes podem enviar mensagens indefinidamente e, portanto, não é possível determinar se ainda terão mais símbolos de entrada para ler. Por esta razão, o protocolo definido no XMLaw assume que a entrada (representada por eventos XMLaw) é potencialmente infinita e portanto, não é possível determinar quando terminará. Para determinar quando o protocolo pára, o XMLaw pára automaticamente quando se atinge um estado final, e a saída é determinada pelo tipo do estado final, que pode ser de sucesso (*success*) ou de falha (*failure*).

ATRIBUTOS

name: String[0..1]

O nome do protocolo

ASSOCIAÇÕES COM OUTROS ELEMENTOS DO XMLAW

initialState: State[1]

Estado inicial. Cada protocolo possui somente um estado inicial.

currentStates: State[1..]*

Conjunto de estados atuais. Devido ao não-determinismo, um protocolo pode possuir um conjunto de possíveis estados atuais durante a sua execução.

É importante perceber que o protocolo não possui associações com os estados finais. Isto ocorre porque os estados possuem uma lista de transições que se originam nele. Logo, para o protocolo, basta conhecer o estado inicial e perguntar ao estado inicial, quais são estados alcançáveis.

EXEMPLO

O Código 15 mostra um exemplo de protocolo cujo atributo *name* é *ping-pong-protocol*. Já o Código 16 mostra o mesmo protocolo sem o atributo *name* declarado.

```
scene01{
    creator{buyer}
    participant{seller, (s0,s1)}
    participant{buyer, (s1)}

    ping-pong-protocol{
        ping-message{ping, pong, inform(ping)}
        pong-message{pong, ping, inform(ping-pong)}

        s0{initial}
        s2{success}

        t1{s0->s1, ping-message}
        t2{s1->s2, pong-message}
    }
}
```

Código 15 – Exemplo de Protocolo com o Nome Declarado

```
scene01{
    creator{buyer}
    participant{seller, (s0,s1)}
    participant{buyer, (s1)}

    ping-message{ping, pong, inform(ping)}
    pong-message{pong, ping, inform(ping-pong)}

    s0{initial}
    s2{success}

    t1{s0->s1, ping-message}
    t2{s1->s2, pong-message}
}
```

Código 16 – Exemplo de Protocolo sem a Declaração do Nome

2.1.11.State

Um estado modela um possível passo ou estágio na evolução da interação entre os agentes. Os estados podem representar situações dinâmicas ou estáticas tais como “esperando pela resposta do comprador” ou “decidindo sobre uma proposta”.

ATRIBUTOS

type: {initial, success, failure, execution}

Se o tipo for *initial*, significa que é o estado inicial do protocolo;

Estados de sucesso (*success*) significam que o protocolo pára com sucesso quando este estado é alcançado;

Estados de falha (*failure*) significam que o protocolo pára com falha quando este estado é alcançado;

Estados de execução (*execution*) quando alcançados não param a execução do protocolo.

label: String

Um nome amigável para o estado

ASSOCIAÇÕES COM OUTROS ELEMENTOS DO XMLAW

outgoingTransitions: Transition[0..]*

Lista de transições que se originam deste estado.

EVENTOS GERADOS

failure_state_reached

Evento gerado quando um estado de falha é alcançado;

success_state_reached

Evento gerado quando um estado de sucesso é alcançado.

EXEMPLO

O Código 16 mostrado anteriormente especifica a declaração de dois estados: s0 e s2. Nota-se que o estado s1, embora seja utilizado pelas transições, não é declarado. Isto ocorre porque estados do tipo *execution* não precisam ser declarados.

2.1.12.Transition

Uma transição é um arco direcionado entre um estado fonte e um estado destino. Ela representa a mudança, causada por um evento XMLaw, entre duas situações diferentes durante a interação.

ATRIBUTOS

event: Event

O evento que pode disparar esta transição.

ASSOCIAÇÕES COM OUTROS ELEMENTOS DO XMLAW

from: State[1]

Estado origem.

to: State[1]

Estado destino.

requiredNorms: Norm[0..]*

Normas que precisam estar ativas para que a transição dispare.

constraints: Constraint[0..]*

Constraints que irão ser verificadas antes que a transição dispare.

EVENTOS GERADOS

transition_activation

Evento gerado quando a transição é disparada

EXEMPLO

O Código 16 mostrado anteriormente apresenta a declaração de duas transições. A transição t1 vai do estado fonte s0 para o estado destino s1, através de um evento do tipo *message_arrival* gerado pela mensagem *ping-message*.

2.1.13.Action

Actions são códigos escritos em Java que são disparados em uma lei em XMLaw. As *actions* podem ser utilizadas para utilizar serviços disponíveis pelo ambiente. Por exemplo, uma *action* poderia invocar um serviço de débito de um banco para cobrar automaticamente a compra de um item durante uma negociação. Neste caso, especifica-se no XMLaw que existe uma classe Java que é capaz de realizar o débito e também o momento em que esta classe deve ser executada.

ATRIBUTOS

class: String[1]

Nome da classe que implementa a *action*.

ASSOCIAÇÕES COM OUTROS ELEMENTOS DO XMLAW

activations: Event[0..]*

Lista de eventos que ativam a *action*.

EVENTOS GERADOS

action_activation

Evento gerado quando a *action* é ativada

EXEMPLO

A lei do Código 17, mostra na linha 17 a declaração de uma action chamada *switchManager*. Esta action é ativada pelo evento *transition_activation* gerado pela transição t4. O Código 18 mostra um exemplo de implementação da *Action*.

```

12:      t4{s1->s1, transfer, [checkTransfer] }
...
17:      switchManager{ (t4), br.pucrio.SwitchManager}
18:      changeBudget{ (t2,t3), br.pucrio.ChangeBudget}
19:      increaseBudgetProhibition{$budgetOwner, (t3), (t4) }
20: }
```

Código 17 – Exemplo de Lei que Declara uma Action

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

Código 18 – Exemplo de Implementação de Action

2.1.14.Elementos Auxiliares

ASSIGNEE

roleRef: String[1]

Referência ao papel do agente que está recebendo a norma.

PARTICIPANT

agent: Agent[0..1];

O agente que é autorizado a entrar na cena.

role: Role[0..1];

Papel do agente que é autorizado a entrar na cena.

states: State[0..];*

Especifica os estados do protocolo que o agente especificado pode entrar na cena. Se nenhum estado é especificado, então pode-se entrar em qualquer estado.

CREATOR

agent: Agent[0..1];.

O agente que é autorizado a criar a cena.

role: Role[0..1];

Papel do agente que é autorizado a criar a cena.

AGENTROLE

agent: Agent[0..1];.

O agente que é autorizado a enviar/receber a mensagem.

role: Role[0..1];

O papel do agente que envia/recebe a mensagem.

2.2. Modelo de Eventos

Os elementos do modelo conceitual são conectados uns aos outros através de um paradigma baseado em eventos. Por exemplo, a norma pode ser composta com a transição para criar uma situação onde a norma fica ciente da ativação de uma determinada transição para se comportar apropriadamente. O pseudocódigo mostrado no Código 19 ilustra esta idéia. Outros elementos também podem ser compostos para alcançar um comportamento mais complexo. Por exemplo, suponha que um comprador (*buyer*) tenha que cumprir a norma 1 (*norm1*) em no máximo 10 minutos. Desta forma, a norma precisará incorporar alguma noção de tempo. Para alcançar isto, compõem-se a norma com o *clock*. Primeiro o *clock* observa quando uma norma é atribuída ao *buyer*. Depois o *clock* começa a contar dez minutos e quando este tempo passar, o *clock* gera um evento do tipo *clock_tick*. Este evento é escutado por uma outra norma (*norm2*), que então proíbe qualquer interação futura do *buyer* (Código 20).

Neste cenário, o elemento norma não foi originalmente concebido para incorporar a noção de tempo, e nem o *clock* foi concebido para incorporar a noção de normas. O baixo acoplamento entre estes dois elementos por causa da abordagem baseada em eventos leva a um modelo flexível de composição, em particular para composições que não foram antecipadas.

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

Código 19 – Pseudocódigo para a Ativação de uma Norma Devido ao Disparo de uma Transição

```
...
t1(s0,s1, message_arrival(m1) )
clock1{
    start to count when listen(norm_activation(norm1))
```

```

        count until 10 min and generate(clock_tick(clock1))
    }
...
norm1{
    give obligation to buyer when
listen(transition_activation(t1))
}
norm2{
    prohibit all interactions from buyer when
listen(clock_tick(clock1))
}
...

```

Código 20 – Pseudocódigo para Compor uma Norma com um *Clock*.

2.2.1. Definição do Modelo Baseado em Eventos

Todos os elementos do modelo conceitual são capazes de perceber e gerar eventos. Mais precisamente, seja E o conjunto composto pelos seguintes elementos: {Law, Scene, Norm, Clock, Protocol, State, Transition, Action, Constraint, Agent, Message, Role}. e denota um elemento de E , ou seja, $e \in E$. Então cada e é capaz de perceber e gerar eventos.

Conforme pode ser visto na Figura 3, todo e possui o mesmo ciclo de vida básico. É importante perceber que não existem restrições quanto ao tipo de evento que pode ativar um elemento. Esta informação é fornecida na especificação da lei.

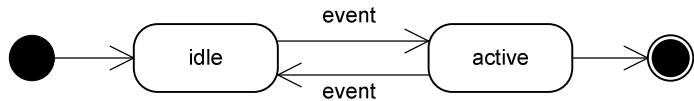


Figura 3 – Ciclo de Vida de um Elemento de Lei

O Código 21 mostra um exemplo onde a especificação do tipo de evento que ativa um elemento, neste caso o elemento *clock*, só é expressa na lei. A linha 16 diz que o *clock* é ativado (vai do estado *idle* para o estado *active*) quando as transições t1 ou t4 dispararem. Este mesmo *clock* é desativado (vai do estado *active* para o estado *idle*) quando as transições t2, t3 ou t4 dispararem. Este mecanismo de eventos relativamente simples permite a composição entre os elementos de forma flexível e desacoplada.

```

...
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)}

```

...

Código 21 – Definição dos Eventos que Ativam um Elemento de Lei

2.3.Contextos

A especificação das leis define vários contextos. Contextos são geralmente hierárquicos e limitam a visibilidade de informações e operações em um determinado sistema. A idéia deste tipo de contexto pode ser explicada fazendo-se uma analogia à estrutura dos sistemas de arquivos. Em um sistema de arquivo, cada diretório pode conter arquivos ou outros diretórios. Desta forma, cada diretório fornece um contexto para os arquivos e diretórios contidos nele.

A especificação de leis de uma organização de agentes pode ser composta pela definição de várias cenas. Essas composições definem contextos, onde os elementos da lei definidos no escopo de uma organização são visíveis para todas as cenas, mas os elementos definidos no contexto de uma cena são somente visíveis na cena em si.

Os eventos gerados no contexto da organização podem ser percebidos em todas as cenas. Os eventos gerados no contexto de uma cena podem ser percebidos tanto na própria cena quanto na organização, mas não são percebidos pelas outras cenas.

Além destes dois contextos (organização e cena), a própria cadeia de propagação de eventos cria um contexto temporário contendo informações com ciclo de vida vinculado a cadeia de propagação. Por exemplo, durante o recebimento de uma mensagem, gera-se um evento de *message_arrival*. Esse evento inicia a cadeia de propagação e eventos e, portanto, um contexto temporário é criado. Este contexto contém as informações da mensagem. Suponha que este evento dispare uma transição, as informações da mensagem estarão disponíveis para a transição, que se disparada, pode adicionar informações extras ao contexto. Quando não houver mais eventos a serem disparados nesta cadeia, o contexto é destruído.

2.4.Lista de eventos

message_arrival
compliant_message

transition_activation
failure_state_reached
success_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
constraint_not_satisfied
action_activation
agent_unavailable

Tabela 3 – Lista de Eventos

2.5.Gramática

Originalmente as leis eram escritas em uma sintaxe baseada em XML. Entretanto, com o objetivo de aumentar a facilidade de escrita do código e a clareza das especificações, foi proposta uma nova sintaxe. O mediador M-Law teve que ser adaptado para dar suporte à nova sintaxe. A gramática da linguagem de representação do modelo conceitual do XMLaw é mostrada no Código 22.

OU
[] opcional
' ' palavra reservada
id = seqüência de caracteres seguidas por letras e dígitos
num = seqüência de dígitos
ε = símbolo vazio
Law = id '{' Scene Action Norm Clock '}'
Scene = id '{' ['time-to-live' '=' num]
Creator
Participant
Protocol
Constraint
Action
Norm
Clock
'}' Scene ε
Creator = 'creator' '{' Agent '}' Creator ε
Agent = '(' Role ',' id ')' Role
Role = id
Participant = 'participant' '{' Agent ',' StateRef '}'
Participant ε

```

StateRef = ElementRef

ElementRef = id |
    '(' id ',' EventType ')'

ListOfElementRef = ElementRef MoreElementRef

MoreElementRef = ',' ListOfElementRef | ε

Protocol = id '{' Message State Transition '}' |
    Message State Transition

Message = id '{' Sender ',' Addressee [ ',' Performative ] |
    ',' Content '}' MoreMessages

MoreMessages = Message | ε

Sender = Agent

Addressee = Agent

Performative = id

Content = id

State = id'{'StateType'}' MoreStates

MoreStates = State | ε

StateType = 'initial' | 'success' | 'failure' | 'execution'

Transition = id '{'
    SourceState '->' DestinationState ',' ,
    Activation
    '}' MoreTransitions
    |
id '{'
    SourceState '->' DestinationState ',' ,
    Activation ',' Conditions
    '}' MoreTransitions

MoreTransitions = Transition | ε

SourceState = id

DestinationState = id

Activation = ElementRef

Conditions = Lists

Lists = '[' ListOfConstaintId ']' |
    '[' ListOfNormId ']' |
    '[' ListOfConstaintId ']' ',' '[' ListOfNormId ']'

ListOfConstaintId = ListOfIds

ListOfNormId = ListOfIds

ListOfIds = id MoreListOfIds

```

```

MoreListOfIds = ',' ListOfIds | ε

Constraint = id '{' JavaClass '}''

Action = id '{' ActivationEvents ',' JavaClass '}''

JavaClass = id

ActivationEvents = Events

DeactivationEvents = Events

Events = '('ElementRef')' |
      '('ListOfElementRef')'

Norm = id '{' NormType ',' Owner ',' ActivationEvents ',' 
      DeactivationEvents '}''

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

Owner = id

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

Time = num [Unit]

Unit = 's' | 'm' | 'h' | 'd'

ClockType = 'periodic' | 'regular'

```

Código 22 – Gramática XMLaw

2.6.Considerações Finais

Neste capítulo, apresentou-se o modelo conceitual do XMLaw e a gramática contendo a sintaxe da linguagem. Durante a apresentação do modelo diversos exemplos foram utilizados para ilustrar os conceitos. Documentou-se em um único lugar os vários aspectos necessários para a especificação de leis com XMLaw. Em comparação com a linguagem em XML original do XMLaw (Paes 2005) a linguagem atual permite uma especificação das leis de maneira mais compacta.

3

Trabalhos Relacionados ao Meta-Modelo

Uma abordagem de governança precisa lidar com dois assuntos importantes: o modelo conceitual (também chamado de linguagem de domínio ou meta-modelo) e mecanismos de implementação que dêem suporte para a especificação e aplicação (*enforcement*) das leis baseadas no modelo conceitual. Nesta seção, discute-se principalmente o primeiro assunto.

Através do modelo conceitual, descrevem-se quais elementos os projetistas podem utilizar para especificar as leis. O modelo especifica o vocabulário e a gramática (regras) que os projetistas podem utilizar para projetar e implementar as leis. O modelo possui um impacto decisivo sobre quanto fácil se torna a especificação e manutenção de leis. É a abordagem utilizada para projetar um software que determina boa parte da complexidade resultante do software. Quando um software se torna muito complexo, é difícil entendê-lo e, consequentemente, mudá-lo (Inc. 2007).

Em 1987, Naftaly Minsky publicou as primeiras idéias sobre leis (Minsky and Rozenshtein 1987). Em 2000, ele publicou um artigo seminal sobre o papel das leis de interação em sistemas distribuídos (Minsky and Ungureanu 2000), que foi chamado de *Law-Governed Interaction* (LGI). Desde então, Minsky tem conduzido o seu trabalho de pesquisa e experimentação baseado nestas idéias (Minsky 2003; Murata and Minsky 2003; Minsky 2005).

Embora o LGI tenha sido utilizado em uma variedade de domínios de aplicação, seu modelo conceitual é composto predominantemente de abstrações relacionadas às informações de baixo nível e sobre questões de comunicação. Exemplos de tais elementos são as primitivas *disconnected*, *reconnected*, *forward*, e o envio e recebimento de mensagens. Embora seja possível especificar regras de interação bastante complexas utilizando estas abstrações de baixo nível, isto pode não ser adequado para projetar ou especificar leis de sistemas considerados complexos. A inadequação ocorre porque é preciso mapear as leis do domínio que estão em um alto nível de abstração para as várias primitivas de baixo nível. Desta

forma, a idéia original da lei é perdida uma vez que ela fica espalhada em várias primitivas de baixo nível. Esta situação reflete o problema de se ter uma linguagem que fornece abstrações que estão muito desconectadas do domínio do problema que se está querendo representar. O desenvolvimento de sistemas complexos e interativos demanda abstrações de alto nível. Sendo assim, as leis devem ser representadas de tal forma que ajudem a reduzir a complexidade com a consequente melhora da produtividade do desenvolvimento de sistemas desta natureza.

As instituições eletrônicas (Esteva 2003) são uma outra abordagem para a utilização de leis de interação. Uma instituição eletrônica (IE) possui um conjunto de abstrações de alto nível que permitem a especificação de leis utilizando conceitos tais como papéis de agentes, normas e cenas. Historicamente, as primeiras idéias apareceram quando os autores analisaram o domínio de um mercado de peixes (Noriega 1997). Os autores perceberam que para obter um nível adequado de regulação sobre as ações dos agentes, o mundo real utiliza instituições que definem um conjunto de regras de comportamento, um conjunto de trabalhadores e um conjunto de observadores que monitoram e garantem o cumprimento das regras. Baseando-se nestas idéias, os autores propuseram um conjunto de abstrações e um software para que os sistemas de software pudessem ser construídos utilizando abstrações similares às do mundo real. Entretanto, embora uma IE forneça abstrações de alto nível, o seu modelo conceitual é bastante inflexível quando considerado sob a ótica de evolução do próprio modelo. Em um campo de pesquisa novo, como é a linha de instituições eletrônicas, a flexibilidade de uma abordagem se torna um fator crítico. As pesquisas nesta área estão em evolução constante e, consequentemente, o modelo que representa as abstrações de leis e suas respectivas implementações também devem ser capazes de evoluir para se adaptar a estas mudanças. Um exemplo de evolução é a utilização de leis de interação para incorporar preocupações de fidedignidade. Nesta situação, em geral, a lei é utilizada para especificar comportamentos indesejáveis e a partir da ocorrência destes comportamentos, especificar um respectivo plano de ação para mitigar as situações indesejadas.

Entretanto, como um modelo conceitual de leis pode evoluir se não se conhece a priori qual a natureza das mudanças? Embora não seja possível prever quais partes irão evoluir, é possível construir o modelo sobre uma base que seja

inerentemente flexível. Os sistemas baseados em eventos tendem a ser flexíveis. Isto ocorre porque nestes tipos de sistema evita-se a dependência direta entre os módulos. Ao invés disso, a dependência ocorre entre os módulos e os eventos que eles produzem ou consomem.

Em geral, o projeto de um software baseado em eventos evita qualquer conexão direta entre o componente responsável por executar uma operação e os componentes responsáveis por decidir quando executá-las. A orientação a eventos leva a um baixo acoplamento entre os módulos e vem conseguindo uma boa aceitação principalmente pelo suporte na construção de projetos flexíveis de software (Meyer 2003).

Em uma arquitetura baseada em eventos, os componentes de software interagem gerando e consumindo eventos. Quando um evento em um componente (chamado de componente fonte) ocorre, todos os outros componentes (destinatários) que declararam um interesse no evento são notificados. Este paradigma dá suporte a uma interação flexível e efetiva entre componentes de software altamente reconfiguráveis (Cugola, Di Nitto et al. 1998) e vem sendo aplicado com êxito em domínios bastante variados tais como interfaces gráficas, sistemas distribuídos complexos (Cugola, Di Nitto et al. 1998), sistemas baseados em componentes (Almeida, Perkusich et al. 2006) e integração de software (Meier and Cahill 2005). Muitas destas abordagens usam sistemas baseados em eventos para gerenciar as mudanças que não podem ser antecipadas durante o projeto do software (Batista and Rodriguez 2000; Almeida, Perkusich et al. 2006). Estas mudanças geralmente são motivadas por um melhor entendimento do domínio e por fatores externos tais como fatores estratégicos, políticos ou orçamentários.

Neste capítulo, destacam-se duas características do modelo conceitual de leis proposto nesta tese. A primeira refere-se ao fato dele ser composto de abstrações de alto nível e a segunda diz respeito ao paradigma orientado a eventos que faz parte do modelo conceitual. O objetivo não é convencer o leitor de que as abstrações propostas no modelo conceitual são melhores que as abstrações propostas em outras abordagens. Ao invés disso argumenta-se que as abstrações de alto nível propostas, tornam possível a especificação de leis consideradas complexas. E com o fato do modelo ser baseado em eventos, novos elementos podem ser adicionados ao modelo facilitando a sua evolução.

No modelo, cada elemento é capaz de ouvir e gerar eventos. Por exemplo, se o modelo possui a noção de normas, então este elemento, a norma, deve gerar eventos que potencialmente podem interessar a outros elementos. Exemplos destes eventos são eventos relacionados ao ciclo de vida, a ativação da norma, a aplicação de sanções especificadas na norma, dentre outros. A norma também é capaz de escutar por eventos gerados por outros elementos do modelo conceitual e então reagir apropriadamente. Por exemplo, se no modelo conceitual existe um elemento que representa a noção de tempo, tal como um relógio, então as normas podem escutar notificações do relógio e então o seu comportamento pode se tornar sensível a variações do tempo. Este mecanismo leva a relacionamentos entre os elementos de lei bastante flexíveis e expressivos. Além disso, se existe a necessidade de se introduzir um novo elemento no modelo, então a maior parte do esforço fica restrito a conectar este novo elemento aos eventos com que ele precisa reagir e descobrir quais os eventos que este novo elemento precisa propagar.

Esta flexibilidade alcançada através de uma abordagem baseada em eventos em conjunto com abstrações de alto nível não está presente em outras abordagens que também utilizam abstrações de alto nível (Esteva 2003; Dignum, Vazquez-Salceda et al. 2004). As vantagens alcançadas com o uso de eventos como auxílio a modelagem também está presente na abordagem LGI (Minsky and Ungureanu 2000), entretanto as abstrações utilizadas são de baixo nível.

Este capítulo ilustra em detalhes como mapear a linguagem de alto nível proposta nesta tese para a abordagem LGI. Este mapeamento é importante por ilustrar que é possível alcançar os resultados produzidos por LGI sem considerar, até o momento, as dimensões de desempenho e segurança.

3.1. Relação do Modelo Conceitual do XMLaw com uma Abordagem de Baixo Nível Baseada em Eventos

Minsky (Minsky and Ungureanu 2000) propôs um mecanismo de coordenação e controle chamado *Law Governed Interaction* (LGI). Este mecanismo é baseado em dois princípios: a natureza local das leis e a descentralização do *enforcement*, ou seja, a descentralização da maneira que se verifica e garante que a lei está sendo cumprida. A natureza local das leis significa que as leis podem regular somente eventos locais em cada *home agent*. Um *home*

agent é um agente que está sendo regulado pelas leis. A descentralização do *enforcement* é uma decisão arquitetural considerada como necessária para alcançar escalabilidade e evitar a presença de um único ponto de falha no sistema.

LGI possui um rico conjunto de eventos que podem ser monitorados em cada controlador. Um controlador é a denominação dada aos agentes responsáveis por realizar o *enforcement*. Cada *home agent* está associado a um controlador. Uma vez que estes eventos são monitorados, é possível utilizar operações para especificar as leis. A união de eventos e operações compõe o modelo conceitual de LGI. Este modelo foi concebido para lidar com decisões arquiteturais para alcançar um alto nível de robustez. Essa decisão levou a um modelo composto por primitivas de baixo nível de abstração. Embora estas primitivas sejam adequadas para muitas classes de problema, algumas vezes é necessário utilizar várias primitivas para alcançar o comportamento desejado. Uma vez que as leis se tornam mais complexas, a manutenção de leis utilizando estas primitivas de baixo nível se torna uma tarefa mais complicada e propensa a falhas.

É possível pensar no LGI como uma máquina virtual altamente escalável cujas instruções são compostas por elementos de lei de baixo nível. Desta forma, seria possível, por exemplo, utilizar as abstrações de alto nível do XMLaw para especificar as leis e em um segundo passo, mapear a especificação para executar sobre a arquitetura LGI. Com o intuito de ilustrar esta idéia, mostra-se como alguns dos elementos que compõem o modelo conceitual do XMLaw podem ser mapeados para várias primitivas do LGI. Embora esta ilustração não seja exaustiva, ela pode ser facilmente estendida para abordar todos os elementos do modelo do XMLaw e a arquitetura LGI. Ainda para facilitar o entendimento e a comparação entre o XMLaw e o LGI, resumiu-se os principais eventos e operações do LGI na Tabela 4 e Tabela 5.

adopted	Representa o momento que um agente adota uma lei LGI e passa a interagir sob suas condições.
arrived	Ocorre quando uma mensagem M enviada por um agente X para o agente Y chega ao controlador de Y. O evento é gerado no controlador do agente Y.
disconnected	Este evento ocorre quando um agente se desconecta de seu

	controlador.
exception	Este evento pode ocorrer uma operação invocada por um agente falha.
obligationDue	Este evento funciona como um relógio que lembra ao controlador que uma obrigação previamente estabelecida expirou. Obrigações são declaradas pela primitiva imposeObligation.
reconnected	Ocorre quando um agente que se desconectou previamente, se reconectou.
sent	Ocorre quando o agente X envia uma mensagem para um agente Y. O evento ocorre no controlador do agente X.
stateChanged	Ocorre quando uma obrigação de estado expira.

Tabela 4 – Lista dos Principais Eventos do LGI

Deliver	Esta operação envia uma mensagem. A estrutura desta operação é dada por: deliver([x,L'],m,y), onde x é o agente remetente, m a mensagem, L' a lei e y o agente destinatário.
Forward	Redireciona uma mensagem.
Add	Adiciona termos ao CS (Control-State).
Remove	Remove termos do CS.
Replace	Substitui termos do CS.
Incr	Esta operação possui a forma incr(f,d) e incrementa o termo $f(n)$ de d unidades.
Decr	Esta operação é a contraparte da operação Incr.
replaceCS	Esta operação possui a forma: replaceCS(termList) e substitui todo o conteúdo do CS com a lista de termos especificada.
addCS	Esta operação possui a forma: addCS(termList). Adiciona a lista de termos ao CS.
imposeObligation	imposeObligation(oType, dt, timeUnit). Declara que uma obrigação de um determinado tipo expira

	após o intervalo de tempo especificado.
repealObligation	repealObligation(oType). Remove todas as operações do tipo especificado.
imposeStateObligation	imposeStateObligation(termList). Faz com que seja gerado um evento stateChanged quando ocorrer qualquer mudança em algum dos termos do CS.
repealStateObligation	repealStateObligation(all). Contraparte da operação imposeStateObligation.

Tabela 5 – Lista das Principais Operações do LGI

A maioria dos eventos listados na Tabela 4 estão relacionados a informações de baixo nível sobre comunicação (*disconnected*, *reconnected*), envio ou recebimento de mensagens (*sent*, *arrived*), ou mudanças de estado no controlador de estados (*stateChanged*). Do ponto de vista das operações, elas também estão relacionadas principalmente a instruções de baixo nível tais como *forward*, *deliver*, *add* e assim por diante.

O protocolo mostrado na Figura 4 pode ser especificado de forma direta em XMLaw através dos elementos de lei *Protocol*, *State*, *Transition* e *Message*. O Código 23 ilustra como este protocolo é especificado em XMLaw. Para alcançar o mesmo comportamento no LGI, pode-se escrever as leis conforme ilustrado no Código 24. Na lei em LGI, foi necessário introduzir dois novos termos que não fazem parte dos termos pré-definidos do LGI: *currentState* e *event*. O termo *currentState* modela os estados atuais do protocolo e o termo *event* simula a geração de eventos. Por exemplo, na primeira linha da listagem de Código 24, um agente *A* envia a mensagem *m1* para o agente *B*. Se o estado atual é *s0*, então o estado *s0* é removido da lista de estados atuais e o estado *s1* é adicionado a esta lista. Após isto, simula-se a geração do evento *transition_activation* e finalmente a mensagem é redirecionada.

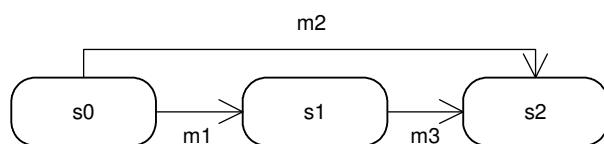


Figura 4 – Exemplo de Protocolo

```
t1{s0->s1, m1}
t2{s0->s2, m2}
t1{s1->s2, m3}
```

Código 23 – Mapeamento do Protocolo da Figura 4 usando XMLaw

```
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).
```

Código 24 – Mapeamento do Protocolo da Figura 4 usando LGI

Os exemplos mostrados na Tabela 6 mostra como situações encontradas no XMLaw podem ser mapeadas para a abordagem LGI. Para isto incorpora-se na semântica expressa em prolog do LGI os termos *event*, *currentState* e *norm*.

1. No recebimento de uma mensagem *m1*, um clock precisa ser ativado para disparar um evento em 5 segundos. O clock deve ser desativado quando a mensagem *m2* chegar.

<pre>XMLaw myXMLawClock{5000,regular, (m1), (m2)}</pre>
<pre>LGI arrived(X, m1, Y) :- imposeObligation("myLGIClock",5). arrived(X, m2, Y) :- repealObligation("myLGIClock").</pre>

2. Disparar a transição *t2* quando um clock gerar um evento do tipo *clock_tick*. A transição muda o protocolo do estado *s1* para o estado *s2*.

<pre>XMLaw t2{s1->s2, myXMLawClock}</pre>
<pre>LGI obligationDue("myLGIClock ") :- currentState(s1)@CS, do(remove(currentState(s1))), do(add(currentState(s2))), do(add(event(t2,transition_activation))), do(forward). /* comentário: o evento obligationDue ocorre quando o tempo especificado a obrigação expirar. */</pre>

3. Declarar um clock do tipo periódico que precisa gerar um evento a cada cinco segundos. Este clock deve ser ativado pela chegada da mensagem *m1* e desativado pela chegada da mensagem *m2*.

<pre> 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). /* comentários: o evento obligationDue ocorre quando o tempo especificado na obrigação expira. Pode-se utilizar um loop na especificação para simular clocks periódicos. Neste exemplo, declara-se um clock e quando o clock expira, um evento obligationDue é gerado o que, por sua vez, ativa outra imposeObligation, e assim por diante. */ </pre>
<p>4. Uma mensagem <i>m1</i> ativa a transição <i>t1</i>. A transição <i>t1</i> muda o estado do protocolo de <i>s1</i> para <i>s2</i>. Uma norma <i>n1</i> precisa ser ativada quando a transição <i>t1</i> disparar. A norma é dada para o agente que recebeu a mensagem <i>m1</i>.</p>
<pre> XMLaw t1{s1->s2, m1} n1{\$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(n1,norm_activation))), do(add(norm(n1,active,valid))). /* comments: Neste caso, o imposeStateObligation causará um evento stateChanged todas as vezes que o termo event(t1,transition_activation) for adicionado ao CS. Então, no terceiro comando, quando este evento stateChanged ocorrer, a norma n1 é ativada. */ </pre>

Tabela 6 – Mapeamento entre o XMLaw e o LGI

3.1.1. Propriedades Globais

De acordo com (Minsky 2005): “qualquer política que pode ser implementada através de um mediador central e que mantenha o estado global da interação de toda a comunidade de agentes, também pode ser implementado via uma lei LGI”. Como um exemplo de propriedade global, suponha a situação da Figura 5. Neste exemplo, 3 agentes estão interagindo no contexto de um determinado protocolo. O agente A envia a mensagem *m1* para o agente B. Como os agentes A e B estão interagindo, seus controladores atualizam o estado atual (cada controlador atualiza o seu próprio estado atual de forma independente). Entretanto, o agente C não participou desta interação e, portanto, seu controlador

não atualizou o estado atual. Isto leva a uma inconsistência entre os estados atuais dos controladores dos agentes A e B e o estado atual do agente C. A causa principal disto é o monitoramento sendo realizado de forma descentralizada e sem nenhuma sincronização explícita.

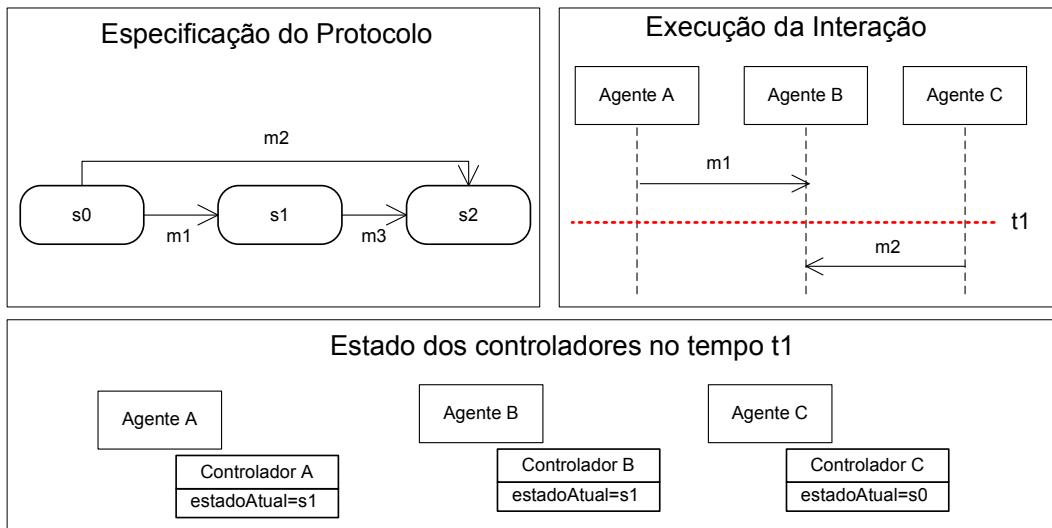


Figura 5 – Exemplo da Necessidade de Sincronização entre os Controladores para a Identificação de Propriedades Globais.

A forma geral para superação deste problema é a utilização de protocolos de sincronização como o *token ring* que foi utilizado na abordagem *Islander* (Esteva 2003). No LGI, este comportamento pode ser alcançado introduzindo-se um coordenador central que recebe todas as mensagens e mantém um estado global consistente. O Código 25 ilustra em LGI como este coordenador (mediador) pode ser especificado na lei. As leis em XMLaw são especificadas de um ponto de vista global (Paes, Carvalho et al. 2007). A abordagem ilustrada no Código 25 possui uma vantagem sobre o mediador centralizado utilizado no XMLaw. Como os mediadores do XMLaw precisam receber todas as mensagens, interpretá-las e só então, eventualmente, descartá-las, eles não podem se proteger contra o congestionamento de mensagens geradas por agentes super-ativos. Isto pode levar o mediador a problemas de negação de serviço (*deny of service*). Por outro lado, utilizando-se LGI, é possível especificar o limite da frequênciа de mensagens emitidas pelos participantes. E o mais importante é que este limite é verificado localmente nos controladores.

```

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)).

```

Código 25 – Redirecionamento de Mensagens para um Mediador Central

3.2.Relação do Modelo Conceitual do XMLaw com uma Abordagem de Alto Nível não Baseada em Eventos

As Instituições Eletrônicas (IE) são uma tecnologia para monitorar e garantir o cumprimento de leis em uma sociedade de agentes. Vários estudos de caso foram apresentados utilizando-se esta abordagem. Eles incluem um sistema para um mercado de peixes (Cuni, Esteva et al. 2004), um ambiente de computação em grade (Ashri, Payne et al. 2006), e uma aplicação para controle de tráfego terrestre (Bou, López-Sánchez et al. 2006).

As IEs utilizam um conjunto de conceitos que possuem pontos de contato com os conceitos utilizados no XMLaw. Por exemplo, tanto as cenas e protocolos do IE especificam, como no XMLaw, a interação de um ponto de vista global. O aspecto temporal é representado através de *timeouts*. Os *timeouts* permitem que sejam disparadas transições depois que um certo número de unidades de tempo tenha passado desde que um estado do protocolo tenha sido alcançado. Por outro lado, por causa do modelo de eventos, o elemento *clock* (relógio) proposto pelo XMLaw pode tanto ativar quanto desativar não apenas transições, mas também outros *clocks* e normas. A conexão de *clocks* e normas permite um comportamento normativo mais expressivo uma vez que as normas se tornam elementos sensíveis ao tempo. Além disso, outra diferença entre os dois modelos é a presença do elemento *action* no XMLaw. Este elemento permite a execução de código Java em resposta a alguma situação na interação.

A Tabela 7 compara as abstrações utilizadas no modelo conceitual de ambas as abordagens. O objetivo desta comparação é relacionar precisamente o XMLaw

com uma abordagem existente e amplamente conhecida na área de governança. Através desta comparação é possível analisar as vantagens e desvantagens de cada uma das abordagens quando se considera o uso delas na implementação de governança. A comparação mostra que, embora as abordagens compartilhem um número razoável de conceitos, eles possuem diferenças importantes tal como, por exemplo, a noção de normas que está presente em IE é melhor definida do que no XMLaw. Por outro lado, o conceito de *actions* do XMLaw pode ser bastante útil para tornar o comportamento das leis mais ativo e integrado com serviços providos pelo ambiente. Essencialmente, a maior diferença entre os dois modelos é a maneira pela qual as abstrações se relacionam para compor uma lei.

Na abordagem de IE existe um conjunto fixo de relacionamento entre os elementos e a maneira pela qual estes elementos são utilizados em conjunto é definida a priori. Um bom exemplo desta idéia é o conceito de *timeout* presente em uma IE. Esta abstração é bastante similar ao *clock* do XMLaw. Entretanto, um *timeout* só pode ser utilizado com transições. Se o modelo de comunicação entre os elementos fosse mais flexível, seria possível, por exemplo, fazer o mesmo que o XMLaw faz e conectar o *timeout* a uma norma.

Instituição Eletrônica	XMLaw	Comentários
Illocutory formulas	Message	Possuem estruturas diferentes, mas possuem propósito similar.
EI vocabulário (ontologia)	Define-se o vocabulário na própria definição da mensagem ao invés de separadamente.	IE define uma ontologia explicitamente. No XMLaw esta definição não é realizada de forma explícita.
Papéis Internos	Não considerado	Papéis internos definem um conjunto de papéis que irão ser desempenhados por “agentes internos”. Uma IE delega a implementação de alguns serviços para estes agentes internos. Um agente externo não pode desempenhar um papel interno.
Papéis Externos	Papel (Role)	
Relacionamento entre papéis	Não considerado	
Controle sobre quem desempenha um papel	Controle sobre quem desempenha um papel	Ambas as abordagens fornecem mecanismos de controle sobre o número mínimo e máximo de agentes que podem desempenhar papéis em uma cena.
Cena (Scene)	Cena (Scene)	As duas abordagens possuem o conceito de Scene. Em uma IE, é necessário especificar quando os agentes entram e saem das cenas. No XMLaw só é necessário especificar o momento de entrada.
Performative Structure	Não considerado	É um tipo especial de cena (Scene) que aceita transições de outras cenas e possui transições para cenas. Eles

		possibilitam a especificação da sequência de execução esperada entre as cenas. No XMLaw, alcança-se efeito similar através de normas: ao término de uma cena, uma norma pode ser ativada e verificada como pré-requisito para iniciar uma cena.
Protocolo (Protocol)	Protocolo (Protocol)	
Estado (State)	Estado (State)	Estados são bastante similares, exceto pelo fato de que no XMLaw existem dois tipos de estados finais: <i>failure</i> e <i>success</i> .
Arco direcionado (Directed edge)	Transition .	Arcos direcionados de uma IE podem ser ativados por uma <i>Illocutory formula</i> , uma <i>timeout</i> ou <i>constraints</i> . No XMLaw transições podem ser ativadas por qualquer evento.
Constraint	Constraint	Em uma IE as <i>constraints</i> são implementadas como expressões booleanas utilizando-se a forma (op expr1 expr2). No XMLaw, elas são implementadas utilizando-se código Java.
Time-out	Clock	Time-outs permitem o disparo de transições após um determinado período de tempo. Por outro lado, um <i>clock</i> é um elemento de propósito mais geral que também pode ser utilizado para disparar transições. Mas ele pode ser utilizado para outros fins tal como fazer com que uma norma expire após um determinado período.
Regras normativas (Normative rules)	Normas (Norms)	Ambas as abordagens possuem as noções de obrigação, permissão e proibição. As regras normativas de uma IE verificam: “QUANDO uma mensagem é enviada SE a mensagem satisfaz determinadas condições ENTÃO outra mensagem com determinadas condições precisa ser emitida no futuro”. Por outro lado, a norma no XMLaw pode ser utilizada para evitar o disparo de transições, ativar ações ou qualquer outro elemento do XMLaw.
Não considerado	Actions	As <i>actions</i> podem ser utilizadas para “plugar” serviços no mediador. Elas podem ser ativadas por qualquer evento do XMLaw e são implementadas através de código Java.
Não considerado	Law	No XMLaw, o elemento <i>Law</i> é o contexto global onde se compartilha informações entre cenas, normas, clocks e actions.

Tabela 7 – Relacionamento entre os Modelos Conceituais de XMLaw e IE

3.3.Moise +

O modelo Moise+ estabelece quais os componentes que formam uma organização. Este modelo foi desenvolvido com o objetivo principal de auxiliar o processo de reorganização. O Moise+ é composto por três dimensões: a estrutura (papéis), o funcionamento (planos globais) e as normas (obrigações) da organização. O aspecto estrutural atém-se aos componentes elementares da organização (papéis) e como estão relacionados (ligações entre papéis, grupos de papéis, hierarquias). O aspecto funcional especifica como os objetivos globais

podem ser atingidos (planos globais, missões e metas). Por fim, o aspecto deôntrico relaciona os dois anteriores, indicando quais as responsabilidades dos papéis nos planos globais. Os principais elementos que compõem o seu modelo conceitual estão descritos na Tabela 8.

Papel	É um tipo primitivo sobre o qual são estabelecidas restrições estruturais e funcionais.
Missão	É parte de um esquema social e, como tal, é uma estrutura de decomposição de metas através de planos. A relação de obrigação e permissão também existe, não é definida na própria missão, mas na sua relação com a estrutura social.
Ligações	São restrições sobre os papéis.
Grupo	Conjunto de papéis, ligações e subgrupos com as cardinalidades.
Instanciação	É um conjunto de pares (agente, papel), um conjunto de grupos instanciados e um conjunto de ligações instanciadas.

Tabela 8 – Elementos do Moise+

O Moise+ é composto por abstrações de alto nível e possui associado um mecanismo de implementação que permite a verificação da conformidade com as leis (Hübner, Sichman et al. 2006). O Moise+ também possui a noção de eventos, entretanto, os eventos são utilizados para comunicar mudanças organizacionais aos agentes. Os eventos não são utilizados para realizar a composição entre os elementos do modelo conceitual, como ocorre no XMLaw.

3.4. Estudos de Caso

Nesta seção, foram escolhidos dois exemplos publicados na literatura para ilustrar a aplicabilidade do modelo do XMLaw e facilitar a comparação com os trabalhos relacionados. O primeiro exemplo já foi publicado utilizando-se a abordagem LGI e o segundo exemplo foi publicado utilizando-se a abordagem IE. Estes dois exemplos foram escolhidos propositadamente para facilitar a análise dos prós e contra das diferentes abordagens.

3.4.1. Estudo de Caso 1: Equipe de Compradores

Este estudo de caso foi publicado em (Minsky and Murata 2004). Para esta tese, a descrição do problema foi ligeiramente modificada, eliminando, por exemplo, a necessidade de uma autoridade de certificação. O exemplo é descrito a seguir:

“Considere-se uma loja de departamento que possui uma equipe de agentes cujo objetivo é prover a loja com as mercadorias que ela precisa. Esta equipe consiste de um gerente e um conjunto de empregados (ou agentes de software que os representam) que estão autorizados como compradores e possuem um orçamento próprio para as compras. Sob circunstâncias normais, a operação da equipe será bem sucedida se todos os membros seguirem a seguinte política:”

1. A equipe de compradores é inicialmente gerenciada por um agente especial chamando *firstMgr*. Mas qualquer gerente desta equipe pode, a qualquer momento, escolher outro agente autenticado como um empregado como o seu sucessor. Uma vez feita a escolha, o gerente perde todos os seus poderes gerenciais e o escolhido os adquire;
2. Caso o agente possua recursos suficientes, é permitido que um comprador emita ordens de compra (POs). O custo da ordem será subtraído do orçamento do agente que emitiu a ordem. A cópia de cada PO emitida precisa ser enviada para o gerente da equipe;
3. Qualquer empregado pode receber um orçamento do gerente. Além disso, cada empregado também pode doar parte do seu orçamento para outros empregados, recursivamente. O gerente pode reduzir o orçamento de qualquer empregado *e*. Como consequência, o orçamento do empregado *e* é bloqueado, ou seja, ninguém pode doar orçamento para ele enquanto o gerente não mudar.

Mensagens. O item 1 da política acima é realizado quando o agente desempenhando o papel de gerente envia a mensagem *transfer* para o empregado que irá ser o sucessor. O item 2 acontece quando o comprador envia a mensagem *purchaseOrder(Amount)*, onde *Amount* é o valor da ordem de compra que irá ser reduzido do orçamento do comprador. Em relação ao item 2, um agente atribui um orçamento para outros através da

mensagem *giveBudget(Amount)*. Então, o concedente do orçamento terá o seu orçamento reduzido pelo valor de *Amount* e o destinatário terá o seu orçamento aumentado do mesmo valor. Finalmente, os gerentes podem enviar a mensagem *removeBudget(Amount)*. O efeito desta mensagem é reduzir o valor de *Amount* do orçamento do destinatário.

Solução XMLaw. XMLaw possui abstrações para decompor o problema em informações menores e mais gerenciáveis. Além disso, também é possível estruturar os etapas de interação de um protocolo de conversação complexo. Neste exemplo, as interações não seguem uma ordem pré-definida e o protocolo não é complexo o suficiente para justificar uma decomposição em várias partes menores. Sendo assim, especificou-se as leis utilizando apenas uma cena que encapsula o protocolo de interação e um conjunto de normas, *actions* e *constraints*. A especificação completa pode ser vista no Código 26 e o código para as *actions* e *constraints* utilizadas na especificação podem ser vistas nas listagens de código: Código 27, Código 28, Código 29 e Código 30.

A explicação da solução começa por uma descrição sucinta da sintaxe e da dinâmica dos elementos. Na Figura 6, também apresenta-se uma notação gráfica do protocolo baseado nos diagramas de estado da UML (Group 2007).

Existem cinco tipos de mensagens que podem ser trocadas pelos agentes, estas mensagens estão especificadas das linhas 02 até as linhas 06. O formato da mensagem é *message-id{sender,receiver,content}*. O símbolo * denota qualquer valor. Também é possível manipular variáveis. As variáveis são armazenadas no contexto. Cada cena possui o seu próprio contexto e também existe um contexto global da lei. Os contextos formam uma hierarquia.

A mensagem especificada na linha 02 significa que a identificação do agente remetente será atribuída à variável *budgetOwner*, o destinatário pode ser qualquer agente e o conteúdo tem a forma *purchaseOrder(valor)*, onde o valor será atribuído a variável *amount*. As mensagens são usadas para ativar cinco transições do protocolo (linhas 09 até 13). Entretanto, as transições t1, t2, t3, e t4 possuem *constraints* que serão verificadas antes de serem disparadas. As transições só disparam se as *constraints* forem satisfeitas. As *constraints* estão

especificadas nas linhas 14 e 15. Uma vez que as transições disparem, algumas delas podem ativar *actions*, conforme pode ser visto nas linhas 16 até 18. Finalmente a transição t2 precisa que a norma especificada na linha 19 esteja ativa para que a transição dispare.

A transição *t1* controla as ordens de compra descritas no item 02 da política. Para ser ativada, a *constraint enoughMoney* referenciada na linha 09 verifica se o remetente identificado pela variável *budgetOwner* (linha 02) possui dinheiro suficiente para emitir a ordem (Código 29). A transição *t2* controla o item 03 da política. Ela se refere a situação onde um empregado ou gerente atribui um orçamento a outro empregado. Então, a constraint *enoughMoney* referenciada na transição *t2* verifica se o remetente possui dinheiro suficiente para efetivar a transferência (Código 29). Depois disso, verifica-se se o empregado que está prestes a receber o dinheiro tem permissão para recebê-lo, conforme especificado na política 03. Esta verificação é feita através da norma *increaseBudgetProhibition* (linha 19). Esta norma é atribuída a um empregado quando o gerente envia a mensagem *removeBudget* e esta mensagem ativa a transição *t3*. No XMLaw, isto pode ser visto na linha 19 onde *t3* é a transição que ativa a norma e *t4* é a transição que a desativa. Portanto, se a norma está ativa, a transição *t2* não é disparada. Se o agente não possui esta norma ativa associada a ela, então a transição *t2* irá disparar. Uma vez que a transição *t2* dispare, a *action changeBudget* é ativada (linha 18). O código desta *action* pode ser visto no Código 30. A transição *t4* é ativada quando o gerente envia a mensagem *transfer* para um empregado. No protocolo, a *constraint checkTransfer* (Código 29) garante que o remetente da mensagem é de fato o gerente. Se a transição *t4* disparar, a *action switchManager* (Código 28) é executada e a norma *increaseBudgetProhibition* é desativada. A *action switchManager* atualiza o gerente atual e uma vez que a norma *increaseBudgetProhibition* não esteja ativada, os empregados que tinham essa norma associada a eles estão agora livres novamente para receber orçamentos através da mensagem *giveBudget*.

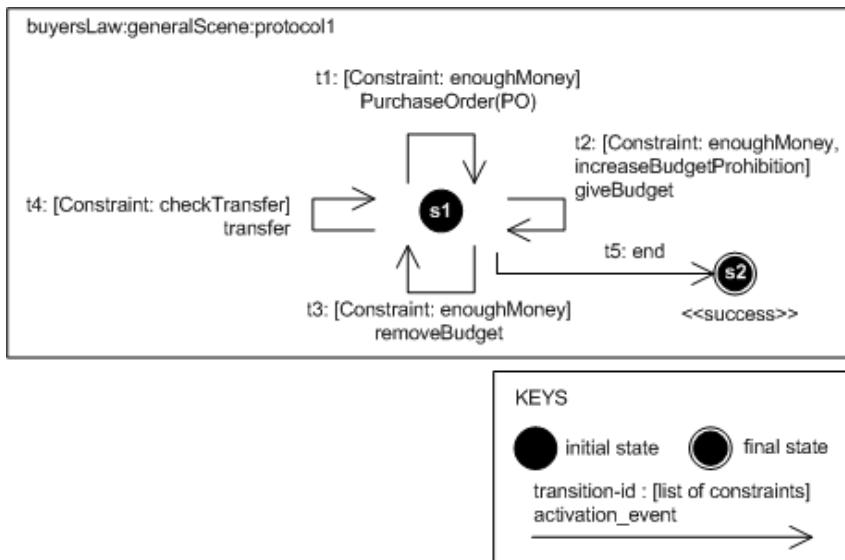


Figura 6 – Protocolo de Interação do Estudo de Caso da Equipe de Compradores

```

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,
11:     [enoughMoney], [increaseBudgetProhibition]}
12:   t3{s1->s1, removeBudget, [enoughMoney]}
13:   t4{s1->s1, transfer, [checkTransfer]}
14:   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: }
```

Código 26 – XMLaw do Estudo de Caso da Equipe de Compradores

```

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;
    }
}

```

Código 27 – Constraint que Verifica se o Agente é de Fato o Gerente Atual

```

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

```

Código 28 – Action que Troca o Gerente Atual

```

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;
        }
    }
}

```

Código 29 – Constraint que Verifica se o Agente que está Cedendo Dinheiro Realmente tem Dinheiro para Ceder

```

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);  
    }  
}  
}
```

Código 30 – Action que Atualiza os Orçamentos e que é Utilizado nas Mensagens de *giveBudget* e *removeBudget*

Discussão. A parte mais importante deste estudo de caso é o Código 26. Este código contém os elementos do modelo conceitual do XMLaw. O código apresentado é predominantemente declarativo e é composto de abstrações de alto nível tais como protocolos de interação, *actions*, *constraints* e normas. Foi possível expressar as regras em vinte linhas de comandos que são relativamente simples de entender mesmo para aqueles não estão acostumados com o XMLaw. Mesmo com o código Java necessário para implementar as *actions* e *constraints*, na maior parte do tempo o projetista pode focar na especificação das leis do Código 26 e utilizar as *actions* e *constraints* como componentes para alcançar a funcionalidade desejada. O modelo de comunicação entre os elementos baseado em eventos está presente na maior parte das declarações. Por exemplo, a linha 19 usa a notificação de eventos para especificar que a norma *increaseBudgetProhibition* é ativada pelo evento *transition_activation* gerado pela transição t3. Além disso, ela é desativada pelo evento *transition_activation* gerado pela transição t4. Outro exemplo é a transição t1 na linha 09 que é ativada pelo evento *message_arrival* gerado pela mensagem *PO*. É claro que a sintaxe da linguagem abstrai a maior parte dos detalhes do projetista e permite que o modelo de eventos funcione “por trás das cenas”. Embora este estudo de caso seja relativamente pequeno, ele é bastante útil para tornar as idéias apresentadas neste capítulo mais concretas. O uso de um estudo de caso que já foi publicado e implementado em outra abordagem torna possível uma comparação bastante direta.

Quando comparado a solução publicada em (Minsky and Murata 2004), as leis do XMLaw fornecem um nível maior de abstração quando analisados sob a ótica de descrição do problema até a sua solução. Por exemplo, a restrição descrita no item 2 da política: “Caso o agente possua recursos suficientes, é permitido que

um comprador emita ordens de compra (POs)” é mapeada diretamente para a *constraint enoughMoney* do XMLaw utilizada na linha 09.

3.4.2. Estudo de Caso 2: Centro de Conferências

Este estudo de caso foi implementado utilizando-se Instituições Eletrônicas e foi apresentado em (Rodriguez-Aguilar 2001; Esteva 2003). O exemplo é descrito como se segue:

“Uma conferência ocorre em um centro de conferências. Diferentes atividades ocorrem simultaneamente em diferentes locais protagonizados por pessoas que desempenham papéis diferentes (palestrante, *chair*, organização, etc.). Durante a conferência, as pessoas se orientam pelos seus interesses movendo-se pelos diferentes locais e se engajando em atividades diferentes. O agente pessoal é um agente que está imerso em um ambiente virtual e que é responsável por auxiliar um participante da conferência na procura de informações de interesse e no diálogo com outros agentes de software.”

O exemplo apresentado em (Esteva 2003) estruturou esta aplicação em seis cenas diferentes: *Information Gathering*, *Context*, *Appointment Proposal*, *Appointment Coordination*, *Advertiser*, e *Delivery*. Entretanto, a única cena em que foram apresentados detalhes suficientes foi a *Appointment Proposal*. Desta forma, esta cena será o foco de comparação neste segundo estudo de caso.

Os participantes desta cena são dois agentes pessoais (PRA). O objetivo desta cena é alcançar um consenso sobre um conjunto de tópicos para discutir em um encontro. A cena se desenvolve de acordo com os seguintes passos:

1. Um dos PRAs (PRA1) toma a iniciativa e envia uma proposta contendo um conjunto de tópicos iniciais para marcar um encontro com o outro PRA (PRA2). Esta proposta possui um tempo que define a sua validade.
2. PRA2 avalia a proposta e pode aceitar, rejeitar ou enviar uma contraproposta para PRA1 com o conjunto diferente de tópicos. A contraproposta também possui uma validade.
3. Quando o PRA1 recebe a contraproposta de PRA2, o PRA1 avalia a contra proposta e pode aceitar, rejeitar ou enviar outra

contraproposta para PRA2. Esta fase de negociação termina quando um consenso for alcançado ou quando um deles desistir de negociar.

Como foi dito anteriormente, os PRAs participam de um ambiente virtual representando um participante da conferência. Quando um PRA alcança um consenso sobre um conjunto de tópicos, ele precisa informar o participante.

Solução XMLaw. A Figura 7 mostra uma representação gráfica do protocolo de interação para a cena *Appointment Proposal*. A especificação XMLaw é mostrada no Código 31. Existem três tipos de mensagens: *propose*, *accept*, e *decline* (linhas 02 até 04). Estas mensagens são utilizadas para disparar a maior parte das transições (linhas 08, 09, 10, 11, 13 e 14). A transição t5 é ativada pelo *clock* na linha 16. Este *clock* representa o tempo de validade das propostas. Ele é ativado todas as vezes que as transições t1, t4 ou t6 disparam, e é desativado quando as transições t2, t3, t4 ou t5 dispararem. Este *clock* está configurado para gerar um evento de *clock_tick* 5000 milisegundos após a sua ativação. A linha 15 declara a norma *app-notification*. Esta norma significa que foi alcançado um consenso na negociação e, portanto, os participantes precisam comparecer ao encontro. Esta norma pode, por exemplo, ser utilizada em outras cenas para impedir que agentes que não tenham cumprido a sua obrigação interajam. A norma é dada ao PRA que aceita a proposta dos tópicos. A aceitação ocorre na transição t2.

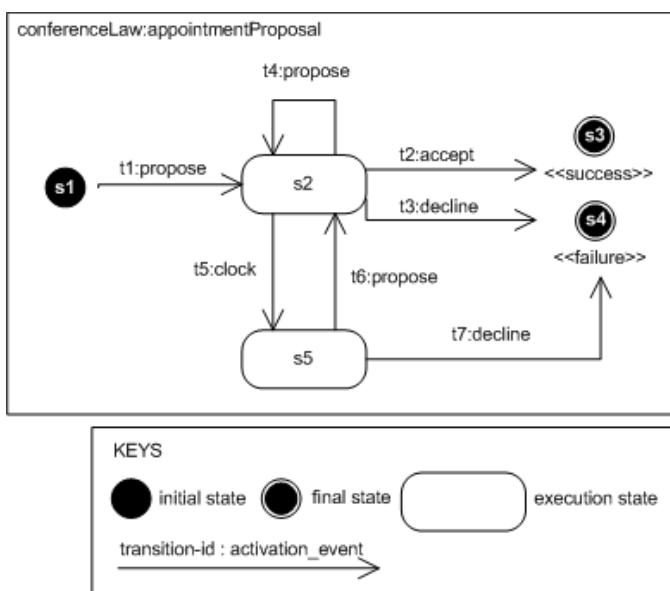


Figura 7 – Protocolo de Interação do Estudo de Caso Centro de Conferências

```

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: }

```

Código 31 –XMLaw da Cena *appointmentProposal*

Discussão. Quanto comparada com a solução apresentada em (Esteva 2003), a solução apresentada aqui possui algumas diferenças importantes: (i) o conjunto de definição de mensagens foi reutilizado muitas vezes na definição do protocolo. Isto levou a um protocolo muito mais simples (por exemplo, o número de transições diminui de 13 para 7); (ii) por causa do modelo baseado em eventos, o *clock* é conectado na lei para disparar as transições. Quando comparado com a IE, a própria transição possui um elemento de *timeout*. Em outras palavras, a transição possui a funcionalidade do *clock*. Esta separação leva a uma melhor separação de conceitos (*concerns*), e a uma melhor reutilização, uma vez que os *clocks* podem ser compostos com outros elementos; (iii) como a norma também está conectada ao evento de modelos, a sua ativação é bem simples, basta que se especifique o evento que ativa a norma.

3.5.Considerações Finais

Neste capítulo, mostrou-se que o modelo conceitual do XMLaw é composto por abstrações de mais alto nível quando comparado com as primitivas do LGI. Também foi mostrado que a noção de eventos leva o XMLaw a um modelo mais flexível para acomodar mudanças e compor os elementos quando comparado com as instituições eletrônicas.

Para ser mais preciso, tanto XMLaw e LGI lidam com a troca de mensagens entre agentes e não são sensíveis ao comportamento interno dos agentes e nem às mudanças no estado interno destes agentes. Em geral, LGI é mais efetivo para leis que são naturalmente locais enquanto o XMLaw é mais efetivo quando as leis são naturalmente globais. Em ambas as abordagens as leis não foram projetadas para especificar todos os detalhes da interação entre os agentes, ao invés disso, as leis especificam as restrições sobre a interação. Do ponto de vista conceitual, o LGI fornece uma abstração do estado de interação de cada um dos agentes (*control state*), um conjunto de eventos relacionados a comunicação e um conjunto de operações para manipular o *control state*. Esse estado funciona basicamente como uma *hashtable* onde os termos são armazenados. Não existe restrição sobre os termos que podem ser utilizados. Esta liberdade pode levar a uma grande flexibilidade que pode ser útil para adaptar a abordagem a diferentes domínios de aplicação. Entretanto, um pequeno conjunto de predicados de alto nível poderia ser mais útil para auxiliar a coordenação e *enforcement* das leis sem a complexidade de ter que criar novos termos. O mapeamento dos elementos do XMLaw para LGI pode ser visto como a tarefa de criar um modelo baseado em Prolog (Clocksin and Mellish 1984; Sterling and Shapiro 1994) do XMLaw. Isto porque os termos do LGI são termos Prolog e como as primitivas do LGI são de baixo nível, pouco poderia ser reutilizado para dar a semântica dos elementos do XMLaw no LGI. O mapeamento pode ser utilizado se existe uma necessidade de uma arquitetura descentralizada, tal como a presente em LGI. Também é importante destacar que embora propriedades globais possam ser implementadas em LGI, se a solução geral apresentada em (Minsky 2005) é usada como ilustrado no Código 25, não estaria sendo feito o uso da descentralização disponibilizada pelo LGI. Por outro lado, também é possível especificar leis que façam uso de informações bem específicas do domínio para sincronizar o estado dos

controladores somente quando for necessário. Entretanto, esta abordagem introduz complexidade para a especificação das leis e ainda traz para a lei conceitos de sincronização de sistemas distribuídos.

O modelo flexível de eventos presente no XMLaw pode contribuir para a construção de modelos de governança de leis mais preparados para acomodar mudanças. Isto é importante, especialmente quando se considera que o próprio XMLaw pode evoluir para incorporar mudanças em prol de expressividade e facilidade de especificação das leis. Um exemplo desta evolução seria a inclusão da noção de normas conforme descrito em (Garcia-Camino, Noriega et al. 2005).

Em resumo, o XMLaw é uma abordagem alternativa às abordagens atuais para a especificação de leis de interação em sistemas multi-agents abertos. As suas principais características são as abstrações de alto nível e um modelo de comunicação entre seus elementos baseado em eventos.

4

Infra-estrutura de Implementação: M-Law

Os avanços da tecnologia de agentes dependem do desenvolvimento de modelos, mecanismos e ferramentas para a construção de sistemas de qualidade. O projeto e a implementação destes sistemas ainda é caro e sujeito a erros. Os frameworks de software lidam com esta complexidade reutilizando projetos e implementações de software validadas na implementação de uma família de aplicações. Um framework orientado a objetos é uma aplicação semi-acabada e reutilizável que pode ser especializada para produzir aplicações customizadas (Fayad, Schmidt et al. 1999).

Neste capítulo, apresenta-se o projeto e a aplicação de um *middleware* para a governança de sistemas multi-agentes chamado M-Law (Paes, Almeida et al. 2004; Paes, Carvalho et al. 2004; Paes, Gatti et al. 2006; Carvalho 2007; Paes, Carvalho et al. 2007). M-Law foi projetado como um *framework* orientado a objetos e os seus *hotspots* permitem conectar infra-estruturas existentes de agentes, mudar o mecanismo de comunicação e adicionar novas funcionalidades. O M-Law garante a regulação das interações dos agentes através dos elementos do modelo conceitual e do modelo de interação baseado em eventos. Este mecanismo intercepta as mensagens enviadas por agentes e verifica sua conformidade com as leis (Figura 8).

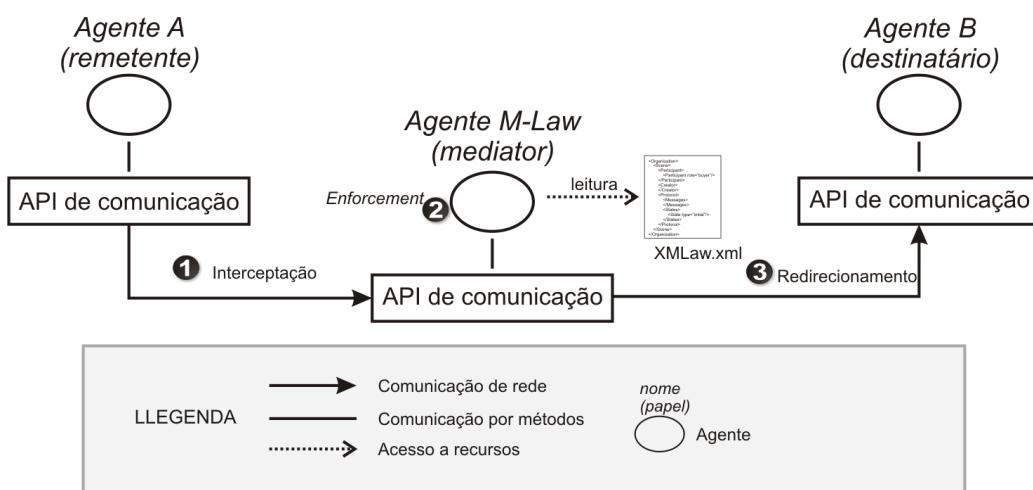


Figura 8 – Arquitetura do Mediador M-Law

Para que o *middleware* M-Law pudesse prover os meios para efetivamente dar suporte à linguagem XMLaw e sua evolução, ele foi desenvolvido com cinco módulos (Figura 9): *communication*, *agent*, *component*, *enforcement* e *event*.

O módulo de *communication* é utilizado para enviar e receber mensagens entre os agentes e o mediador M-Law. O módulo *agent* contém as classes que os desenvolvedores de agentes devem utilizar para implementá-los. Este módulo por sua vez, utiliza o módulo de *communication*. O módulo de *enforcement* é utilizado para mapear elementos de lei descritos usando a linguagem XMLaw para o modelo de execução interno ao M-Law. O módulo *event* é responsável pela notificação e propagação de eventos tão importantes para a realização do modelo de eventos previsto no modelo conceitual de XMLaw. O módulo *component* define um conjunto de classes abstratas e concretas, além de interfaces, que permitem que novas funcionalidades sejam inseridas. De uma forma geral, este módulo contém os componentes que implementam o comportamento dos elementos da linguagem XMLaw.

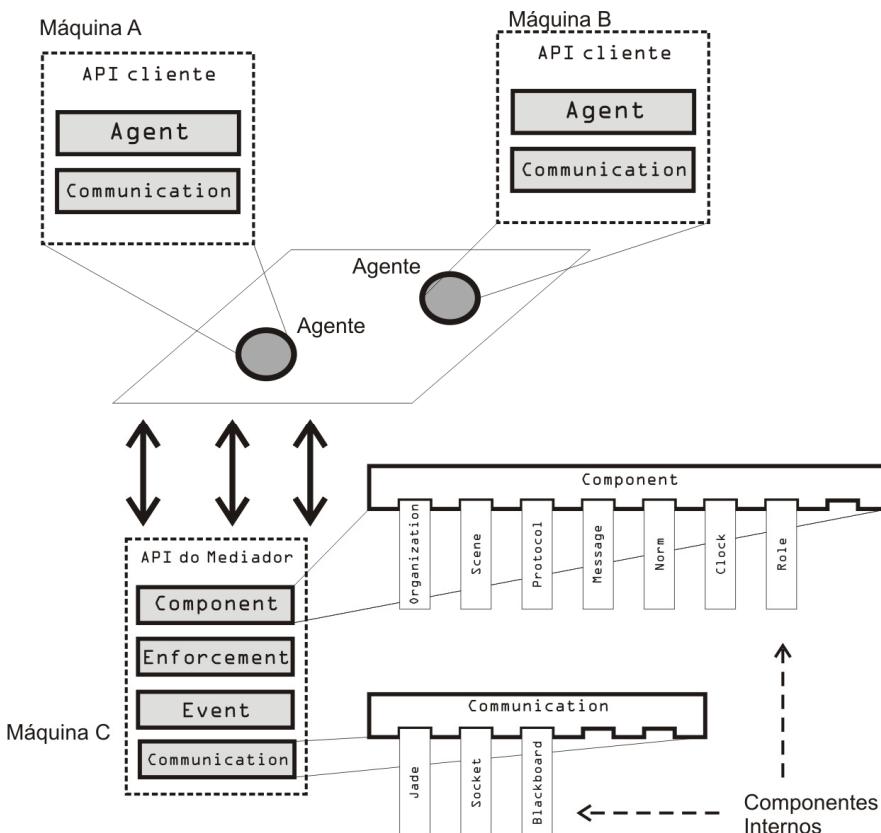


Figura 9 – Componentes Principais do M-Law

Um agente cliente desta solução utiliza os módulos *agent* e *communication*. A máquina de interpretação e *enforcement* de leis, chamada de Mediador, engloba

os outros três módulos, *enforcement*, *event* e *component*, e também utiliza o módulo *communication*. Os módulos de *enforcement*, *event* e *component* não são visíveis para os desenvolvedores de agentes, mas podem ser estendidos para evoluir as funcionalidades do Mediador.

Elementos como cenas, relógios e normas são implementados e associados ao módulo *component*. O módulo *component* contém um conjunto de classes que o mediador implementa para representar o comportamento dos elementos da linguagem XMLaw. Por exemplo, a Figura 10 detalha o elemento Scene (cena) em XMLaw e o conjunto de classes que implementam o seu comportamento. Na Figura 10, o elemento Scene em XMLaw é mapeado para hierarquias distintas, porém relacionadas, de elementos de descrição (*IDescriptor*) e elementos de execução (*IExecution*). Estas hierarquias são pontos de alteração previstos no módulo Componente, e elas são usadas para a proposição de novos elementos na definição da lei, permitindo, por exemplo, que o modelo conceitual de XMLaw evolua com mais facilidade.

As classes e interfaces principais dos módulos *component* e *enforcement* fornecem a estrutura que deve ser implementada pelos elementos da linguagem. Esta estrutura está resumida adiante:

Handlers (SimpleHandler e ComposedHandler) – O módulo Interpretador tem um interpretador XML padrão que lê uma especificação de lei em XML e delega o tratamento de cada tag a um tratador específico. Este tratador é responsável por receber os tokens identificados pelo interpretador e deve construir os descritores de elementos.

IDescriptor – Ele representa o modelo de objeto de uma *tag* XML. Por exemplo, na Figura 10, a *tag* scene do XML é representada pela classe SceneDescriptor. A sua maior responsabilidade é criar instâncias de elementos de execução (*IExecution*) a partir do descritor, através do método *createExecution()*. Esta interface é definida no módulo *component*.

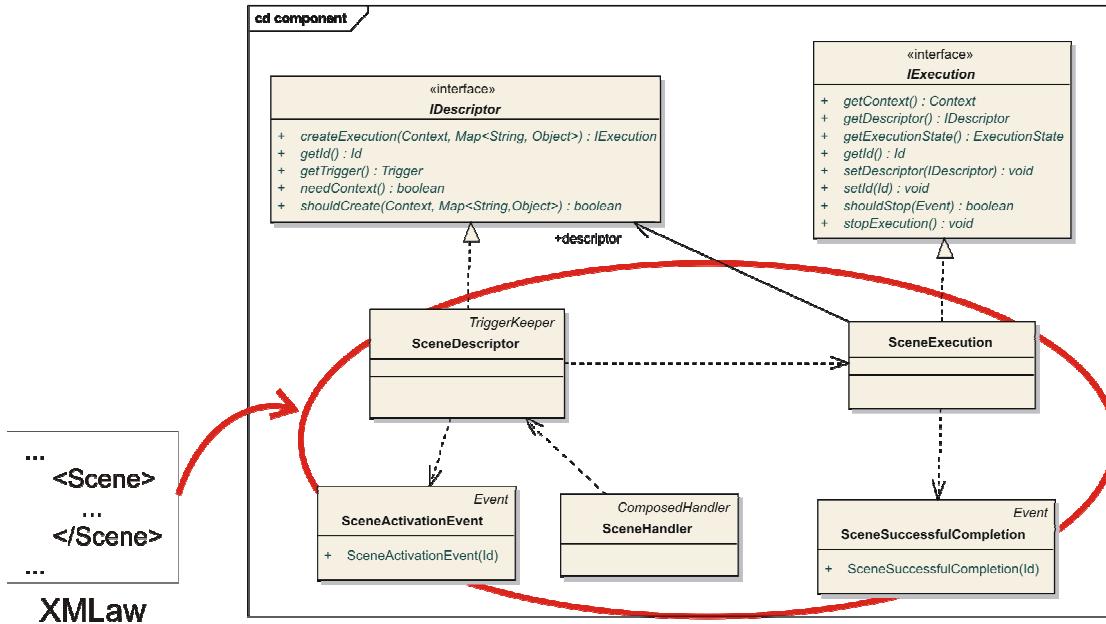


Figura 10 – Projeto do Elemento Scene

IExecution – Um objeto que implementa a interface **IExecution** é uma instância de um elemento representado pelo objeto **IDescriptor**. Por exemplo, uma cena pode ser instanciada muitas vezes e podem existir diferentes instâncias executando ao mesmo tempo (vários leilões executando em paralelo, por exemplo). Cada instância (**IExecution**) deve manter os seus atributos e controlar o seu ciclo de vida. A interface **IExecution** define todas as operações de *callback* necessárias pelo módulo Componente para controlar as suas instâncias.

Para utilizar o M-Law é necessário a execução de quatro passos principais. Primeiro, é preciso escrever as leis utilizando a linguagem XMLaw. Depois o mediador precisa ser iniciado através da execução dos arquivos de script fornecidos com o M-Law. O terceiro passo é indicar ao mediador a localização do arquivo de lei. Finalmente o quarto passo é iniciar os agentes da aplicação.

Desenvolvedores de agentes podem querer estender a classe **Agent** oferecida pela API cliente M-Law. Esta classe fornece métodos para enviar e receber mensagens e métodos para a comunicação direta com o mediador. O mediador pode fornecer informações úteis sobre o estado corrente de uma interação, por exemplo: as cenas que estão executando, quantos agentes estão participando destas cenas, dentre outras. Na verdade, a classe **LawFacade** fornece métodos para a comunicação direta com o mediador, e a classe **Agent** fornece métodos para o envio e recebimento de mensagens. A Figura 11 apresenta estas classes.

Os desenvolvedores de agentes são encorajados a utilizar a classe Agent por herança ou por delegação. Entretanto, é possível ainda que estes desenvolvedores construam os seus próprios agentes utilizando as tecnologias ou arquiteturas de sua preferência. O único requisito que um agente deve satisfazer é como se comunicar utilizando mensagens FIPA-Agent Communication Language (Fipa 2002) com o mediador M-Law. A implementação padrão de M-Law utiliza Jade (Bellifemine, Poggi et al. 1999) como ambiente de comunicação e desenvolvimento básico de agentes de software.

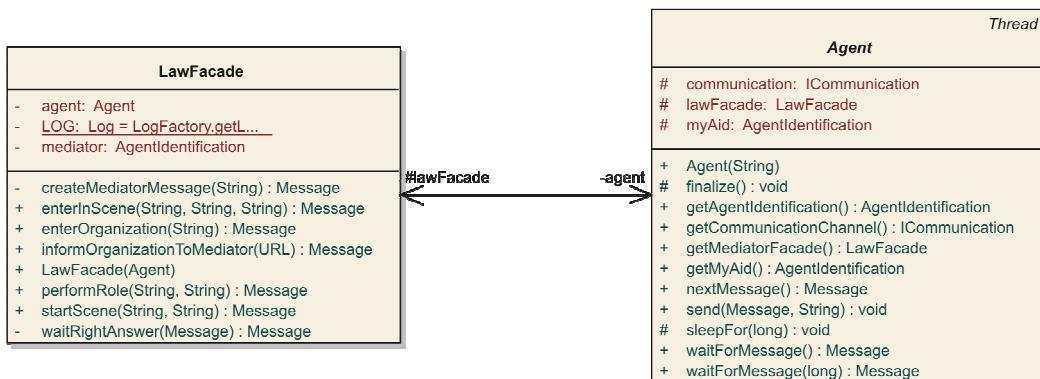


Figura 11 – API cliente: classes LawFacade e Agent

4.1.Ciclo de Vida do Mediador MLaw

Esta seção tem como objetivo explicar as alterações necessárias no mediador M-Law para interpretar leis de interação com pontos de extensão de interação. Conforme explicado, o mediador M-Law é a estrutura de software utilizada para interpretar as leis de interação e monitorar a conformidade dos agentes com o comportamento desejado para o sistema.

O ciclo de vida previsto para o mediador pode ser consultado na figura (Figura 12). Durante a fase de configuração do mediador no estado OCIOSO, ele recebe as leis de interação que vigorarão a partir do início do monitoramento. Esta lei de interação é interpretada pelo componente *enforcement*. Este interpretador verifica se a implementação da lei XMLLaw está bem formada e é consistente. É responsabilidade do interpretador mapear os elementos descritos na lei, para uma estrutura própria do modelo de execução que será utilizado em tempo de execução, e este processo não admite má-formação dos elementos.

Em razão da proposta de pontos de extensão de interação (Carvalho 2007), este processo ocorrerá em dois passos (INTERPRETANDO e ESTENDENDO).

Primeiramente, a lei base será verificada e sua estrutura de execução será criada. Caso ela esteja com alguma má formação, o mediador irá para o estado LEI BASE INCONSISTENTE. Depois de fornecer um conjunto de leis bem formadas, as extensões propostas serão interpretadas e todos os pontos de extensão de interação devem ser refinados e materializados em elementos concretos.

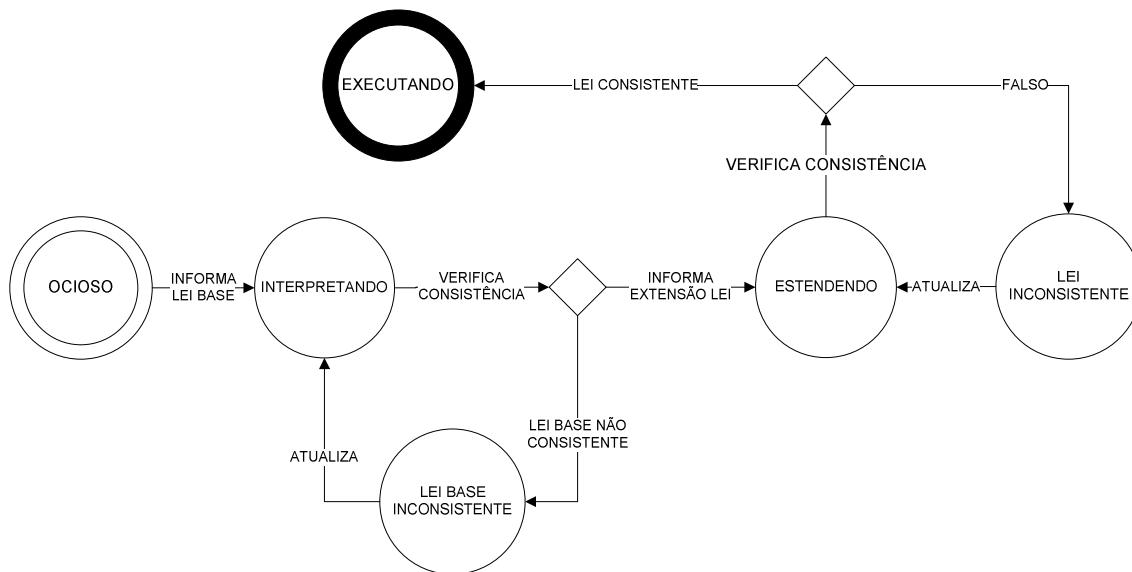


Figura 12 – Ciclo de Vida do Mediador M-Law

Estes dois passos permitem a verificação de algumas regras de construção. Por exemplo, se a lei é definida como concreta, ela não pode deixar nenhum ponto de extensão em aberto. Isto é, a lei deve estar completamente definida. Todos os elementos precisam estar implementados, caso contrário o interpretador indicará um erro e irá para o estado LEI INCONSISTENTE. Caso a lei esteja inconsistente, será solicitada uma nova extensão. No final do processo, com a lei consistente o mediador é capaz de executar e mediar a conversação dos agentes (estado EXECUTANDO).

Para a realização deste ciclo de vida, o mediador M-Law foi projetado de forma a separar classes de definição e descrição de leis (Descriptor) e classes de execução e controle do estado corrente da lei (Execution). Esta separação já foi explicada na Figura 10.

4.2.Arquitetura e Implementação

O M-Law é composto por sete pacotes principais, conforme pode ser visto na Figura 13. Alguns destes pacotes são utilizados pelo mediador, enquanto outros

podem ser utilizados pelos agentes da aplicação. A Figura 14 mostra como esta divisão ocorre. No restante desta seção, cada um dos pacotes é descrito e, quando necessário, mostra-se um diagrama de classes contendo as principais classes de cada pacote.

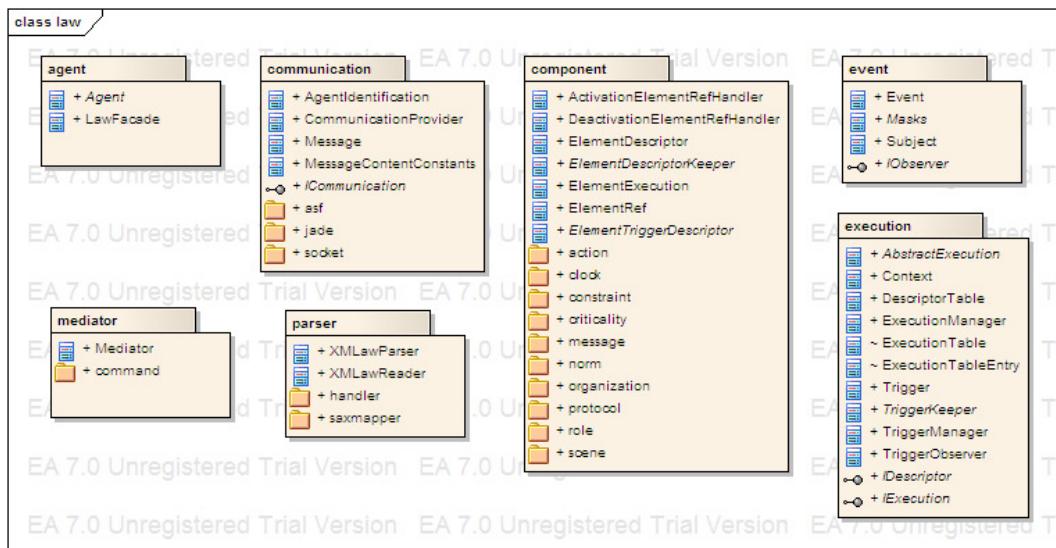


Figura 13 – Diagrama de Pacotes

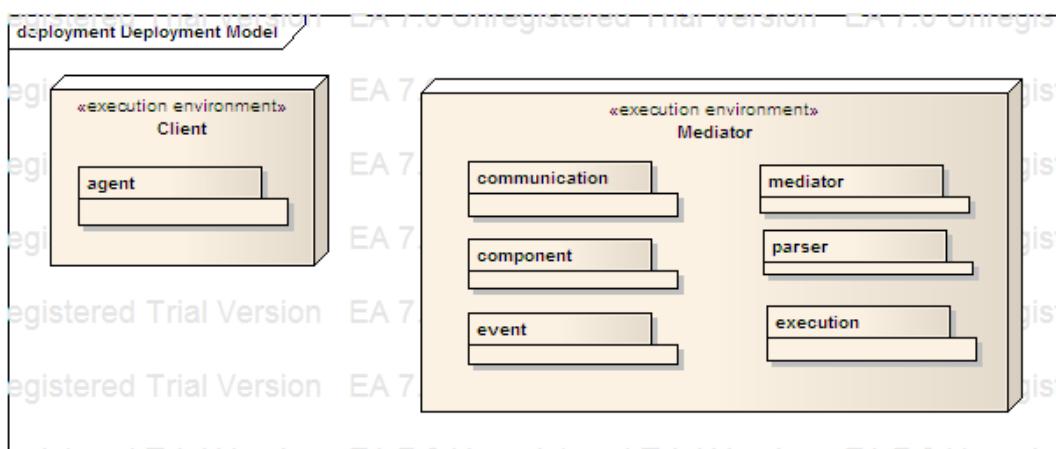


Figura 14 – Diagrama de Deployment

4.2.1. Pacote *agent*

Existem duas maneiras de criar um novo agente: construindo-o desde o início ou usando a classe *Agent* disponibilizada por este módulo. Esta classe fornece métodos que auxiliam no envio e no recebimento de mensagens e é completamente integrada com a abordagem de leis. Caso o agente a ser desenvolvido já herde de alguma outra classe, ainda assim é possível utilizar a

classe *Agent* provida pelo módulo através da técnica de delegação de chamada de métodos e, assim, alcançar os benefícios fornecidos pela classe *Agent*.

O envio e o recebimento de mensagens utilizando a classe *Agent* é realizado com uma chamada de método. A classe *Agent* possui uma instância da classe *ICommunication* como um de seus atributos. Este atributo representa o canal por onde o agente envia e recebe mensagens. Além disso, a classe *Agent* também possui uma referência para a classe *LawFacade*. Esta classe contém métodos para operações de leis, tais como entrar em uma cena e desempenhar um determinado papel. A Figura 15 mostra as classes *Agent* e *LawFacade*.

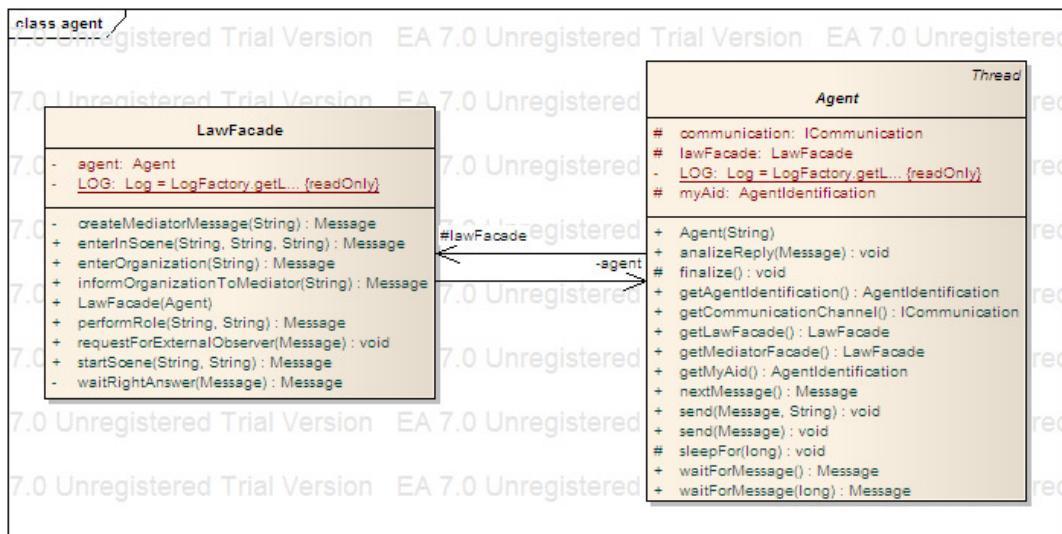


Figura 15 – Diagrama de Classes do Pacote *agent*

4.2.2. Pacote *communication*

Agentes podem utilizar diferentes formas de comunicação. Eles podem utilizar uma infra-estrutura baseada nos padrões SOAP (Box, Ehnebuske et al. 2000), podem implementar infra-estruturas proprietárias utilizando sockets, ou mesmo padrões de comunicação entre agentes tais como os definidos pela FIPA(Fipa 2002). Cada aplicação possui diferentes requisitos de performance, flexibilidade, interoperabilidade, entre outros. Uma camada de comunicação deve se adequar a esses requisitos. Desta forma, a camada de comunicação proposta fornece uma interface que permite que as aplicações mudem a implementação de determinadas partes para que os requisitos específicos de cada aplicação possam ser atendidos.

A interface *ICommunication* (Figura 16) define métodos para o envio e recebimento de mensagens. Por se tratar apenas de uma interface, as

implementações deste método não são fornecidas por *ICommunication* e, portanto, precisam ser providas por implementações da interface para que as funcionalidades sejam fornecidas. O método de envio de mensagem é o método *send(Message msg)*. O comportamento esperado deste método é o envio de uma mensagem ao destinatário especificado na mensagem. Para o recebimento de mensagens, três métodos foram definidos, sendo dois métodos “bloqueantes” e um método “não bloqueante”. O método *waitForMessage():Message* bloqueia a execução do programa que invocou este método até o recebimento de alguma mensagem. Quando uma mensagem for recebida, a chamada é desbloqueada e a mensagem é retornada. Uma versão um pouco modificada deste método é o método *waitForMessage(long milliseconds)*. Este método funciona de forma análoga ao *waitForMessage()* exceto que ele bloqueia o programa que o chamou até que uma mensagem seja recebida ou que o tempo especificado no parâmetro tenha se esgotado. Desta forma, este método retorna a mensagem recebida caso ela tenha chegado ou *null* caso nenhuma mensagem tenha chegado e o tempo tenha expirado. Os métodos *wait* são bloqueantes uma vez que o programa que os invoca têm a sua execução bloqueada. Porém, esta interface define um método denominado *nextMessage():Message*. Este método, ao contrário dos outros dois, não bloqueia a execução, mas retorna a próxima mensagem que ainda não foi lida, ou *null* caso nenhuma mensagem tenha chegado até o momento da invocação do método. Este comportamento do método *nextMessage* sugere que implementadores da interface *ICommunication* implementem uma estrutura de fila para que todas as mensagens recebidas sejam armazenadas e quando o método *nextMessage* for invocado, a primeira mensagem da fila seja retornada.

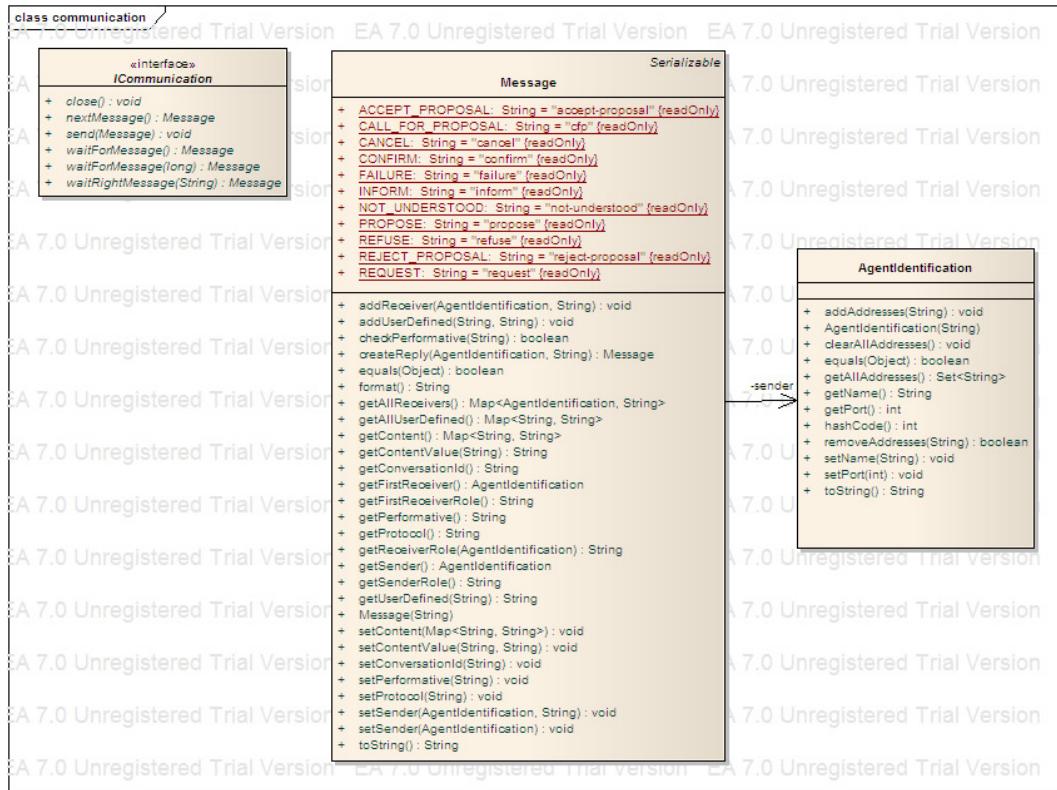


Figura 16 – Diagrama de Classes do Pacote *communication*

A Figura 17, mostra como o framework JADE (Bellifemine, Poggi et al. 1999) foi utilizado para prover as funcionalidades definidas pela interface da camada de comunicação. O JADE implementa um mecanismo de comunicação compatível com os padrões definidos pela FIPA. Este mecanismo é acessível de forma relativamente transparente através da classe *jade.core.Agent*, provida pela implementação JADE. Então, a classe *JadeCommunication* reutiliza a implementação desse mecanismo de comunicação JADE através da delegação dos métodos da interface *ICommunication* para uma instância da classe *jade.core.Agent*.

Também foram realizadas outras implementações da camada de comunicação, sendo que uma utilizou *sockets* e outra o framework ASF (Silva 2004).

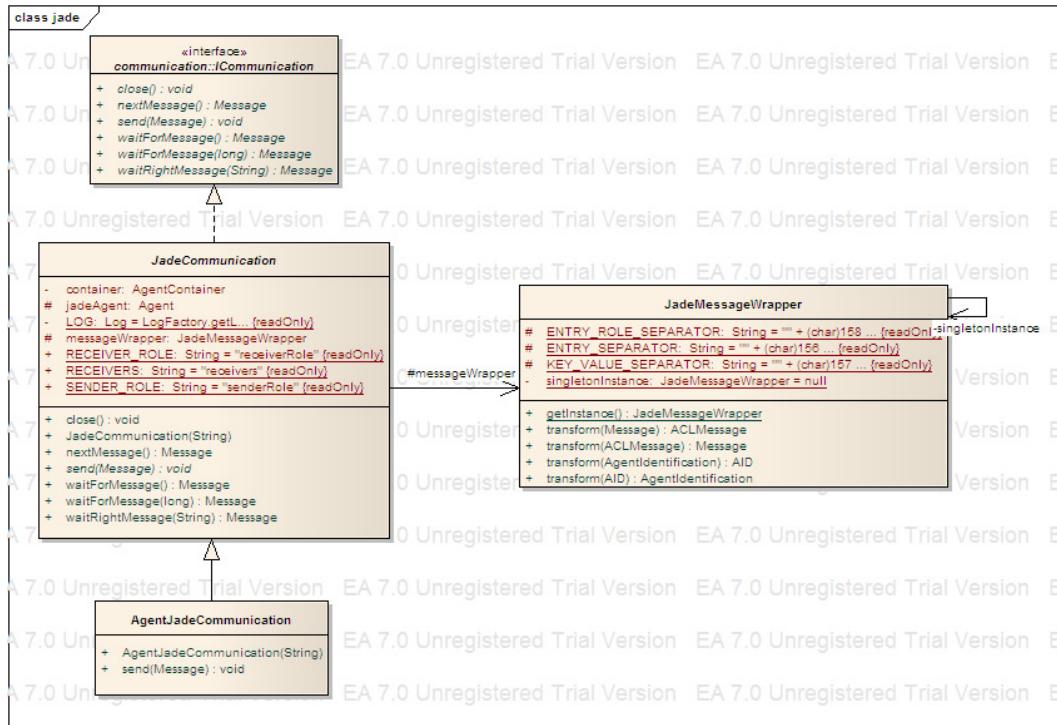


Figura 17 – Diagrama de Classes do Pacote *jade*

4.2.3.Pacote *event*

Este módulo fornece a implementação para que a comunicação entre os elementos do XMLaw seja baseada em eventos. A classe *Event* representa um evento no sistema. Um evento possui principalmente: um identificador único, o identificador do elemento do xmlaw que originou o evento, um conjunto de variáveis que podem ser adicionadas ao evento e o tipo. O evento pode ser de qualquer um dos tipos definidos na classe *Masks*. A estrutura de notificação e subscrição dos eventos é baseada no padrão *Observer* (Gamma, Helm et al. 1995).

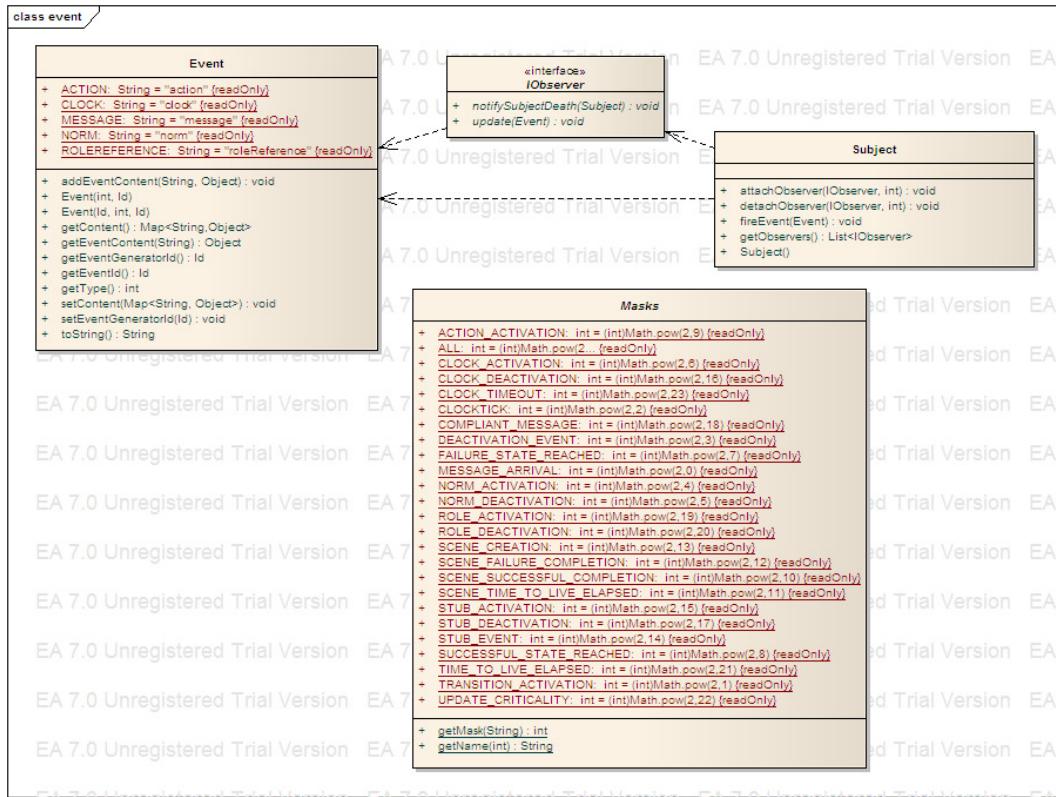


Figura 18 – Diagrama de Classes do Pacote *event*

4.2.4. Pacote *execution*

Este pacote contém as classes que implementam a lógica de execução dos elementos. O diagrama de classes é exibido na Figura 19.

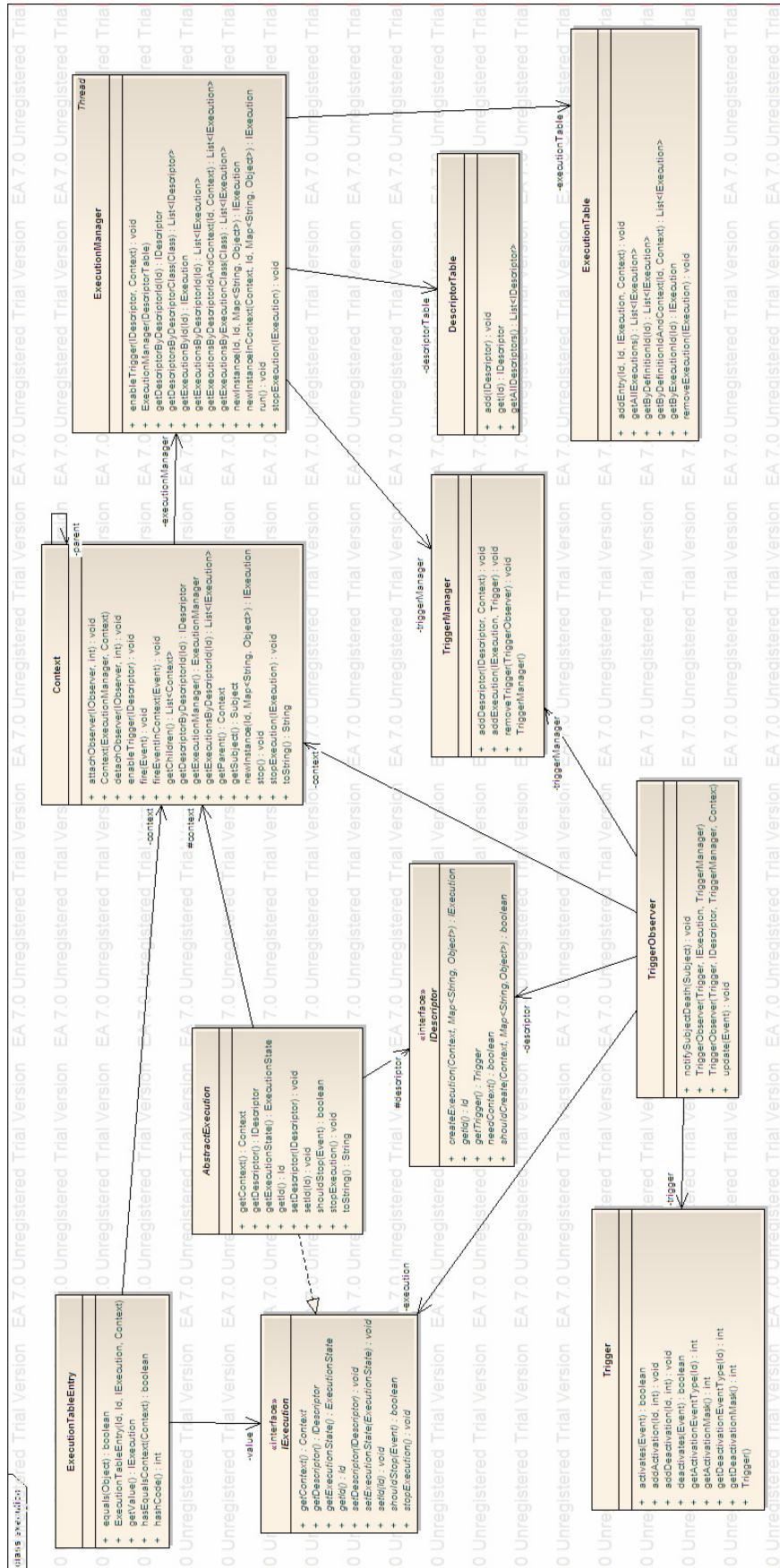


Figura 19 – Diagrama de classes do Pacote *execution*

4.2.5. Pacotes *component*, *mediator* e *parser*

É no pacote *component* que os elementos do XMLaw estão localizados. Cada elemento é um pacote contido dentro do pacote *component*. O pacote *parser* implementa um analisador léxico e sintático para da linguagem XMLaw. Atualmente existem duas implementações deste pacote: uma para a sintaxe em XML e outra para a sintaxe proposta nesta tese. Finalmente, o pacote *mediator* contém o agente mediador. É este agente que recebe as mensagens redirecionadas para ele e inicia a interpretação das mensagens em função da lei.

4.3. Trabalhos relacionados ao Middleware

4.3.1. AMELI

Para dar suporte à especificação de uma instituição eletrônica (Esteva 2003) foi desenvolvido um middleware chamado AMELI (Esteva, Rosell et al. 2004) e um editor gráfico que permite a especificação através da combinação de elementos gráficos e textuais. Além da especificação, este editor também permite a verificação de algumas propriedades, mais especificamente: integridade, que permite verificar as auto-referências entre os elementos da linguagem estão corretas; *liveness*, que permite verificar se algum agente estará bloqueado na *performative structure* e que o estado final das cenas sempre é alcançado, corretude do protocolo; e finalmente a verificação se os agentes podem cumprir as normas especificadas.

O middleware verifica se a especificação da instituição eletrônica está sendo cumprida durante a interação dos agentes. Este mecanismo utiliza o JADE (Bellifemine, Poggi et al. 1999) como camada de comunicação e é composto de 3 elementos principais:

- *Institution Manager* é o elemento responsável por autorizar os agentes a entrar na instituição, iniciar o sistema e controlar a participação dos agentes entre as cenas. Uma vez autorizados, é associado a cada agente um agente *Governor*.
- O *Scene Manager* é responsável por controlar a execução de uma cena, identificando quais os agentes podem participar da cena.

- Cada *Governor* é responsável por mediar a comunicação o agente e o resto da instituição. Eles dão aos agentes toda a informação de que eles precisam para participar de uma instituição.

Um dos pontos importantes que vale ser ressaltado nesta abordagem é que o protocolo é especificado sob um ponto de vista global. Desta forma, é possível saber qual o exato estado de execução deste protocolo. Na abordagem, os protocolos são especificados através de uma máquina de estados finita, onde não existe a idéia de concorrência. Desta forma, os *governors* (responsáveis por atualizar o estado da execução do protocolo ou cena) precisam se coordenar para que somente um deles atualize o estado do protocolo por vez. Além disso, uma vez que a atualização do estado de execução do protocolo é feita, todos os *governors* precisam atualizar a sua visão do estado do protocolo para poder executar alguma mudança de estado. Uma transição que era válida em um estado anterior pode não ser mais válida no novo estado.

Em uma possível solução para esta questão, poderia existir um elemento centralizador que mantém o estado atual da execução de uma cena e, portanto, quando cada *governor* precisar atualizar o estado, é preciso “travar” o estado atual, solicitar uma cópia deste estado, realizar a interpretação da transição de estados, possivelmente modificar o estado e enviar de volta o novo estado destravando-o. Desta forma, a cada mensagem que o agente envia, existem mais duas, uma para a recuperação e outra para a atualização do estado. Somente este fato não ocasiona muitos problemas em termos de ordem de complexidade do algoritmo. Entretanto, introduz-se um único ponto de falha no sistema. Ou seja, caso este elemento que detém as informações sobre o estado da cena se torne indisponível, todos os agentes que participam da cena passam a não poder se comunicar.

Na abordagem, existe um elemento centralizador que mantém o estado atual da execução de uma cena. Porém, várias cópias do estado atual estão distribuídas entre os *governors*. Esta solução elimina o problema de se ter apenas um único ponto de falha, porém, introduz um custo de comunicação relacionado a manter os estados dos *governors* atualizados. Cada mensagem enviada pelos agentes que muda o estado da execução do protocolo tem como consequência a atualização do estado deste protocolo em todos os *governors*. Ou seja, a cada mensagem existe o custo de um *broadcast* para atualização dos estados. Logo, se x mensagens forem

enviadas entre os agentes, o número total de mensagens do sistema é de aproximadamente: $x \times n$, onde n é o número total de agentes em uma cena. Este alto custo de coordenação torna esta solução imprópria para sistemas onde existam um grande número de agentes interagindo em uma mesma cena.

4.3.2.S-Moise+

O S-Moise+ (Hübner 2003; Hübner, Sichman et al. 2006) propõe uma arquitetura (Figura 20) com algumas similaridades em relação à utilizada pelo MLaw. A Tabela 9 compara as duas arquiteturas.

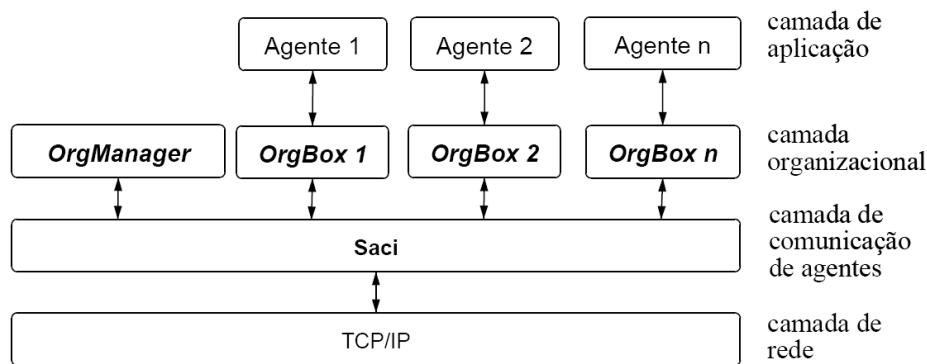


Figura 20 – Arquitetura do S-Moise+

S-Moise+	M-Law	Comentário
OrgBox	Módulo Agent	O OrgBox é uma interface que os agentes utilizam para acessar a organização e os demais agentes. O Módulo Agent no XMLaw possui esta mesma finalidade
Saci	Módulo Communication	Todos os dois elementos fornecem uma abstração para a camada de comunicação. Uma diferença é que o módulo communication do M-Law foi projetado de forma desacoplada do mecanismo de implementação. Atualmente, existe uma implementação do mecanismo de comunicação em JADE, por exemplo, e nada

		impede que esta implementação seja trocada pelo próprio Saci.
OrgManager	Mediador do M-Law	Ambos os elementos possuem a função principal de verificar se a especificação está coerente com a execução.

Tabela 9 – Comparação entre os elementos da arquitetura do M-Law com o S-Moise+

4.4.Considerações Finais e Exemplos de Utilização do M-Law

A primeira versão do M-Law foi criada em 2004 (Paes, Almeida et al. 2004; Paes, Carvalho et al. 2004) e desde então vem sofrendo sucessivas evoluções. Em uma dissertação de mestrado defendida em 2005 (Paes 2005), o M-Law foi detalhadamente documentado. No capítulo 05 desta dissertação, encontra-se um guia de desenvolvimento de agentes utilizando o framework. Optou-se por reapresentar o M-Law como contribuição desta tese porque ele passou por consideráveis modificações, dentre as quais é possível destacar:

- Mudança da interpretador para a nova sintaxe da linguagem – originalmente as leis eram escritas em XML. Entretanto a ausência de uma ferramenta de auxílio a especificações das leis tornava a tarefa de escrever as leis bastante suscetível a erros além de tornar a leitura mais difícil. Assim, optou-se por criar uma nova linguagem de especificação. O impacto desta mudança no framework foi a construção de uma nova camada de *Handlers* e *Descriptors* que reconhecessem a nova sintaxe.
- Adição de novos eventos – a inserção do agente detector de falhas, fez com que o M-Law precisasse reconhecer esse agente como um agente especial e consequentemente gerenciasse toda a geração e propagação de eventos relacionados a ele.

No restante desta seção, mostra-se a utilização do M-Law em vários domínios diferentes.

4.4.1. Protótipo do Sistema Especial de Liquidação e Custódia do Banco Central (SELIC)

O SELIC foi escolhido por se tratar de fato de um sistema distribuído de governança (Carvalho 2007). Atualmente em sua versão de produção, mais de 4000 instituições financeiras utilizam este aplicativo para negociar títulos públicos e solicitar a efetiva liquidação e alteração de custódia. As instituições financeiras implementam a abstração de agentes de software autônomos, que podem entrar e sair do sistema e que se comunicam via troca de mensagens; caracterizando portanto um cenário de sistema aberto. O sistema SELIC foi mapeado diretamente para um sistema multi-agente governado por leis, pois neste contexto existe uma instituição reguladora que funciona como mediadora nas interações de compra e venda de títulos, e também em outras operações relacionadas a instituições financeiras. Em (Carvalho 2007), mostrou-se uma estratégia para a especificações das leis utilizando XMLaw neste domínio, e o M-Law foi utilizado como plataforma de implementação.

4.4.2. Teste de Integração

O trabalho relatado em (Rodrigues, Carvalho et al. 2005) possui como principal foco a estratégia de como escrever casos de teste e receber os resultados da execução destes testes. O sistema é considerado como uma composição de vários subsistemas distribuídos e cada subsistema é visto como um agente. Neste contexto, o XMLaw foi utilizado para especificar o comportamento esperado do sistema e o M-Law foi utilizado para monitorar o comportamento que efetivamente estava ocorrendo. O M-Law foi integrado com outros software que permitiam a escrita dos casos de testes e a geração de relatórios. Os resultados deste experimento ilustraram a capacidade de integração do M-Law com outras soluções de software.

4.4.3. Criticalidade de Agentes

A técnica de replicação de agentes é definida como a ação de criar uma ou mais cópias de um ou mais agentes de um sistema multi-agente. Esta técnica é considerada por muitos como uma maneira de implementar tolerância a faltas em sistemas multi-agentes. O experimento realizado em (Gatti, Lucena et al. 2006) reaproveitou a estrutura de monitoramento do M-Law para identificar a variação

de criticalidade dos agentes. A medida que os agentes iam se tornando críticos, o M-Law envia uma mensagem para uma estrutura de replicação que se encarrega de criar as réplicas. Para que este comportamento fosse alcançado, foi preciso modificar o XMLaw para a inclusão de novos elementos. Estes novos elementos foram adicionados através do módulo *component* do M-Law.

5 Fidedignidade

O objetivo desta seção é apresentar a terminologia comumente utilizada na literatura de tolerância a faltas. Esta introdução é importante para auxiliar na identificação dos aspectos de tolerância a faltas na abordagem apresentada nesta tese.

Um software é dito fidedigno quando se pode, justificavelmente, depender dele assumindo riscos de danos compatíveis com o serviço prestado pelo software (Avizienis, Laprie et al. 2004). Comumente fidedignidade é definida em termos probabilísticos, indicando a probabilidade de que o sistema esteja funcional em um determinado momento. Esta definição deriva conceitos e métricas tais como *mean-time-to-failure* (MTTF). É possível ainda identificar dois focos principais em fidedignidade: o projeto de sistemas fidedignos e a avaliação da fidedignidade. Esta separação é comumente classificada como os atributos de fidedignidade (perspectiva de avaliação) e os meios para alcançá-los (perspectiva de projeto).

Sob a perspectiva de avaliação, a fidedignidade pode ser decomposta nos seguintes atributos (Staa 2006):

Disponibilidade: estar pronto para prestar serviço correto sempre que se necessite do software.

Confiabilidade: habilidade de sempre prestar serviço correto. Conforme observado em (Boehm and Basili 2001), mais de 50% das falhas observadas em sistemas em uso devem-se a erros de operador. Ou seja, confiabilidade não é somente assegurar que o sistema esteja em correspondência exata com a sua especificação.

Segurança (safety): habilidade de evitar consequências catastróficas, ou de grande envergadura, afetando tanto os usuários como o ambiente.

Proteção: habilidade de evitar tentativas de agressão bem sucedidas.

Privacidade: habilidade de proteger dados e código contra acesso indevido.

Integridade: ausência de alterações não permitidas.

Robustez: habilidade de detectar, o mais cedo possível, eventuais falhas de modo que os danos (as correspondentes consequências) possam ser mantidas em um patamar aceitável.

Recuperabilidade: habilidade em ser rapidamente reposto em operação fidedigna após a ocorrência de uma falha.

Manutenibilidade: habilidade de ser modificado (evoluído) ou corrigido sem que novos problemas sejam inseridos e a um custo compatível com o tamanho da alteração.

Depurabilidade: facilidade de diagnosticar e eliminar possíveis causas de problemas a partir de relatos gerados.

Nota-se que todos os atributos de fidedignidade citados acima possuem uma relação direta com avaliação, ou seja, suas definições estão relacionadas com a medição em maior ou menor grau da presença de cada atributo. Por exemplo, a definição de disponibilidade é facilmente associada a uma métrica que indique o quanto disponível o sistema esteve em um determinado intervalo de tempo.

Por outro lado, a perspectiva de projeto de sistemas fidedignos, frequentemente caracterizada como os meios para se alcançar melhores medidas para os atributos, se preocupa com métodos e técnicas que propiciam a construção de sistemas fidedignos. Sob esta ótica é possível destacar quatro categorias (Laprie and Randell 2004):

Prevenção de faltas: objetiva a prevenção da introdução de faltas no sistema. A maneira mais comum de se realizar a prevenção de faltas é através de métodos de controle da qualidade (Team 2006).

Tolerância a faltas: objetiva evitar falhas devido a presença de faltas.

Remoção de faltas: objetiva reduzir o número e a severidade das faltas.

Predição de faltas: objetiva estimar o número de faltas atuais, a evolução deste número de faltas e suas prováveis consequências.

A definição de faltas, falhas, erros encontrada na literatura é muitas vezes vaga e ambígua. Em (Cristian 1991) por exemplo, o autor reconhece que “o que uma pessoa chama de falha, uma segunda chama de falta e uma terceira chama de erro”. O entendimento preciso destas noções evita más interpretações e auxilia no entendimento do escopo das contribuições propostas nesta tese. Por esta razão e baseado no trabalho publicado por Gärtner (Gärtner 1999), a seguir será apresentada a definição destes conceitos.

Um sistema distribuído pode ser modelado como um conjunto finito de processos que se comunicam utilizando um subsistema de comunicação através da troca de mensagens compostas de um alfabeto finito de símbolos. As variáveis de cada processo definem um *estado local*. Cada processo executa um algoritmo local que resulta em uma seqüência de transições no seu estado local. Cada transição de estado define um evento, que pode ser um evento do tipo *send*, um evento do tipo *receive*, ou um evento *interno*.

O estado geral do sistema distribuído é chamado de *configuração*, e é composto pelo conjunto de todos os estados locais de todos os processos mais as mensagens em trânsito. Para auxiliar no entendimento destas definições, considere a listagem de Código 32. Este código descreve um algoritmo distribuído onde dois processos enviam mensagens alternadas um para o outro. O *estado local* do processo Ping é dado pelo valor das variáveis *z* e *ack*. O estado geral do sistema, ou *configuração*, é dado pelo valor de todas as variáveis (*z*, *ack* e *wait*) e pelo conjunto de mensagens que possam estar em trânsito (*a* ou *m*).

Os processos esperam que as condições de guarda (lado esquerdo do símbolo \rightarrow) sejam verdadeiras para que o lado direito seja executado acarretando na mudança de seu estado local e consequentemente na mudança da configuração do sistema.

```

process Ping
    var z: integer init 0
        ack: boolean init true
    begin
         $\neg$ ack  $\wedge$  rcv(m)  $\rightarrow$  ack := true; z := z + 1
        ack            $\rightarrow$  snd(a); ack := false
    end

process Pong
    var wait: boolean init true
    begin
         $\neg$ wait       $\rightarrow$  snd(m); wait := true
        wait  $\wedge$  rcv(a)  $\rightarrow$  wait := false
    end

```

Código 32 – Pseudocódigo do Sistema Distribuído Ping-Pong

A definição de falta utilizada neste trabalho é baseada na observação de que os sistemas mudam suas configurações baseadas em dois tipos de eventos: uma operação normal do sistema ou a ocorrência de faltas. Sendo assim, uma falta pode ser modelada como sendo uma mudança indesejada no estado de um processo. Como um exemplo, a listagem de Código 33 mostra o código do

processo Pong modificado para a inclusão de uma falta. A falta é incluída através da adição da variável *up* com o valor inicial *true* e da ação *up* → *up* := *false*. Como consequência, no momento em que a ação de falta for executada, o processo Pong não enviará mais mensagens e não responderá ao processo Ping.

```
process Pong
var wait: boolean init true
error up: boolean init true
begin
  { * normal program actions * }
  up ∧ ¬wait      → snd(m); wait := true
  up ∧ wait ∧ rcv(a) → wait := false
  { * fault actions * }
  up                  → up := false
end
```

Código 33 – Processo Pong com a Inclusão de uma Falta

A adição da variável *up* e da ação de falta transformou o sistema em um sistema com falta. A inclusão da falta pode levar ao sistema a não comportar de acordo com a sua especificação. Quando isto acontece, utiliza-se o termo falha. Ou seja, uma falha é definida como o fato do sistema não apresentar o comportamento esperado de acordo com a sua especificação.

Desta forma, as quatro principais técnicas para a construção de sistemas fidedignos (prevenção, tolerância, remoção e predição) se preocupam, principalmente, com a investigação das propriedades e técnicas para lidar com sistemas e faltas.

5.1.Detectção de Falhas

Frequentemente em sistemas distribuídos é preciso identificar quais os processos estão operacionais e quais não estão (*crashed*). Entretanto, as várias variáveis envolvidas neste tipo de sistema tais como latência de rede, perda de mensagens e heterogeneidade de processamento tornam esta tarefa não trivial. Os *detectores de falhas* (Tel 2000) são as entidades de software responsáveis por esta tarefa. A principal função de um detector é identificar um determinado conjunto de falhas e comunicar a aplicação sobre a detecção. Os detectores estão presentes em vários dos mais importantes algoritmos de tolerância a faltas em sistemas distribuídos tais como *Leader Election* e *Atomic Commitment* (Tel 2000). Em geral, os detectores são implementados como um conjunto de processos distribuídos que se comunicam para chegar a uma decisão sobre a ocorrência de

falha em um processo do sistema. Existem diferentes implementações de detectores conforme a aplicação, desta forma, os protocolos de comunicação, os tipos de falhas identificadas e os algoritmos para a tomada de decisão são exemplos de pontos de variação.

Uma das principais contribuições dos detectores acontece sob o prisma da Engenharia de Software. De fato, os detectores permitem que as aplicações deleguem os detalhes dos algoritmos de detecção de falhas para os detectores e permaneçam em um nível mais alto de abstração. Desta forma, de maneira análoga às linguagens de programação de alto nível em relação às linguagens de máquina, os detectores podem ser vistos como uma camada de abstração sobre os problemas de detecção de algumas classes de falhas melhorando, assim, o entendimento e a construção de aplicações mais complexas.

6**Incorporação de Fidedignidade na Abordagem de Governança****6.1.Implementação de Duas Estratégias de Tolerância a Faltas Através do XMLaw**

Tem havido uma quantidade considerável de pesquisa usando a noção de leis de interação para definir o comportamento esperado de um sistema multi-agente (Castelfranchi, Dignum et al. 1999; Dignum 2002; Esteva 2003; Felicíssimo, Lucena et al. 2005; Garcia-Camino, Noriega et al. 2005; Minsky 2005; Chopinaud, El Fallah Seghrouchni et al. 2006; Paes, Carvalho et al. 2007). Neste capítulo introduzem-se leis como uma maneira de estruturar o sistema para tolerância a faltas. A idéia principal é a de que os mediadores são instrumentos adequados para a detecção de problemas e para a especificação de estratégias de recuperação uma vez que uma falta tenha sido detectada. As estratégias de detecção são especificadas através das leis.

Este capítulo também discute como alguns atributos de fidedignidade podem ser interpretados em uma especificação de leis e apresenta a especificação de duas técnicas de tolerância a faltas para ilustrar a abordagem proposta.

6.1.1.Leis e Fidedignidade

Quando comparado à outras abordagens, o XMLaw possui algumas características que o tornam adequado para se tratar fidedignidade na especificação das leis. A flexibilidade intrínseca à abordagem orientada a eventos aliada ao alto nível de suas abstrações não está presente em outras abordagens (Minsky and Ungureanu 2000; Esteva 2003; Dignum, Vazquez-Salceda et al. 2004). Esta flexibilidade permite que o modelo conceitual do XMLaw esteja mais preparado para acomodar mudanças. Isto é muito importante especialmente quando se considera a aplicação de leis para a representação de aspectos que originalmente não estavam considerados no projeto de sistemas abertos tais como preocupações sobre fidedignidade.

No capítulo 5, foram descritos os principais conceitos relativos a fidedignidade. Também foram descritos quatro meios para alcançar fidedignidade: prevenção de faltas, tolerância a faltas, remoção de faltas e predição de faltas. Quando se introduz a idéia de utilizar leis para lidar com fidedignidade é importante discutir como estes meios podem ser incorporados em uma especificação de lei. O principal benefício da incorporação destes aspectos nas leis é a reutilização da infra-estrutura necessária para a implementação de leis (mediador e a linguagem). Discute-se, a seguir, como estes aspectos podem ser interpretados do ponto de vista de leis.

Prevenção de faltas – a prevenção de faltas durante o desenvolvimento de um software é o principal objetivo das metodologias de desenvolvimento (ex: modularização, encapsulamento, utilização de linguagens fortemente tipadas, etc.). Melhorias no processo de desenvolvimento também auxiliam a redução do número de faltas em sistemas em produção através da sistemática aplicação de técnicas de qualidade de software (Avizienis, Laprie et al. 2004). Um dos problemas que levam a faltas é a presença de requisitos mal definidos ou ambíguos. A especificação das leis é, na verdade, uma especificação precisa do comportamento esperado do sistema. Esta especificação pode ser utilizada para (i) guiar o desenvolvimento de agentes individuais que compõem o sistema; (ii) guiar o desenvolvimento de scripts de teste relacionados a integração entre os agentes; e (iii) agir como assertivas de execução durante a execução do sistema. Todos estes fatores podem ser incorporados em algum processo de desenvolvimento já existente. Por exemplo, as atividades de um processo poderiam incluir a especificação de casos de uso, especificação das leis, desenvolvimento dos agentes, teste dos agentes utilizando as leis, e assim por diante. Uma lei bem formulada auxilia na prevenção de falhas durante a execução do software.

Tolerância a faltas – As técnicas de tolerância a faltas são compostas de duas fases principais: detecção do erro e recuperação. Os mediadores usados nas abordagens baseadas em leis podem fornecer suporte à detecção de situações de erro. Adotou-se aqui a definição de erro publicada em (Avizienis, Laprie et al. 2004), onde um erro é definido como a parte do estado do sistema que pode levar a uma falha do serviço (Figura 21).



Figura 21 – Relacionamento entre falta, erro e falha (Avizienis, Laprie et al. 2004)

Comumente, os mediadores são implementados como *middlewares* que interceptam a comunicação entre os agentes e agem de acordo com a especificação das leis. Consequentemente, é possível escrever leis que estão preocupadas com a detecção de erros. Por exemplo, no XMLaw algumas possíveis causas de falhas podem ser:

- A própria especificação da lei que pode não representar corretamente o comportamento esperado do sistema, ou seja, o desenvolvedor pode ter escrito uma lei errada. Como consequência, esta lei errada pode levar a falhas no sistema.
- Em XMLaw é possível especificar componentes Java (*actions* e *constraints*) que irão ser invocados pelo mediador de acordo com a lei. Entretanto, estes componentes podem conter faltas, que por sua vez podem levar a falhas.
- A interação entre os agentes pode não ocorrer de acordo com o que foi especificado nas leis. Em alguns casos, a não conformidade com as leis pode significar que o sistema está em um estado de erro como consequência de alguma falta. As leis podem ser utilizadas para detectar e especificar estratégias para lidar com estas situações. O XMLaw possui um conjunto de eventos que podem ser escutados para detectar situações de erro, tais como: (i) *message_not_compliant*. Este evento ocorre quando o mediador recebe uma mensagem de um agente qualquer e esta mensagem não está em conformidade com o padrão de mensagens esperado; (ii) *constraint_not_satisfied*. Este evento ocorre quando uma *constraint* não é satisfeita. Tipicamente, a *constraint* impede o disparo de uma transição. Mais uma vez, a não satisfação da *constraint* pode significar que o sistema está em um estado de erro; (iii) agentes tentando entrar em uma cena sem ter permissão; (iv) quando um *clock* gera um evento do tipo *clock_tick* pode significar que um

determinado agente, que deveria enviar uma mensagem, não estava disponível.

Alem das situações de detecção de erros discutidas acima, também é possível estabelecer estratégias de recuperação através da realização de gerenciamento de erros (*error handling*) ou gerenciamento de faltas (*fault handling*). O estudo de caso apresentado neste capítulo mostra alguns exemplos de estratégias de recuperação implementadas através das leis.

Remoção de faltas – Alguns exemplos de técnicas de remoção de faltas são: inspeções, verificação formal de modelos e testes. Em (Rodrigues, Carvalho et al. 2005), apresentou-se uma abordagem de testes e uma arquitetura para a geração de relatórios de teste utilizando o XMLaw. Uma vez que as leis especificam o comportamento esperado do sistema como um todo, a idéia foi escrever agentes *mock* que implementam o comportamento esperado pelos casos de teste. Os agentes *mock* são análogos aos *mock objects* (Mackinnon, Freeman et al. 2001). Desta forma, os desenvolvedores podem realizar testes de forma incremental, testando os agentes reais enquanto eles interagem com os agentes *mock*.

Prevenção de Faltas – Um dos principais objetivos da prevenção de faltas é a identificação, classificação e avaliação dos eventos que podem levar a falhas no sistema. Em (Gatti, Lucena et al. 2006), o XMLaw foi aplicado para identificar a criticalidade dos agentes em tempo de execução. Quando um agente se torna muito crítico, de forma a prevenir a indisponibilidade dos serviços prestados pelo sistema, as leis especificam ações que interagem com mecanismos de replicação para criar réplicas dos agentes mais críticos.

Como foi discutido anteriormente, as leis e a arquitetura do mediador fornecem uma forma adequada de incorporar os conceitos de fidedignidade. Embora se tenham discutido vários atributos (meios) de fidedignidade, no decorrer deste capítulo o foco da discussão neste capítulo é a apresentação de um estudo de caso com o objetivo de demonstrar o tratamento de tolerância a faltas em XMLaw. A idéia é mostrar que a especificação das leis podem incorporar as técnicas de tolerância a faltas para auxiliar o sistema no oferecimento correto dos seus serviços.

6.1.2. Implementação de Estratégias de Tolerância a Faltas

O estudo de caso utilizado nesta seção é baseado no sistema de controle de vendas apresentado em (Xu, Randell et al. 1995). No estudo de caso, mostra-se como são implementados os requisitos deste sistema. Mais especificamente, apresenta-se como duas estratégias de recuperação ilustradas na Figura 22 podem ser especificadas através das leis.

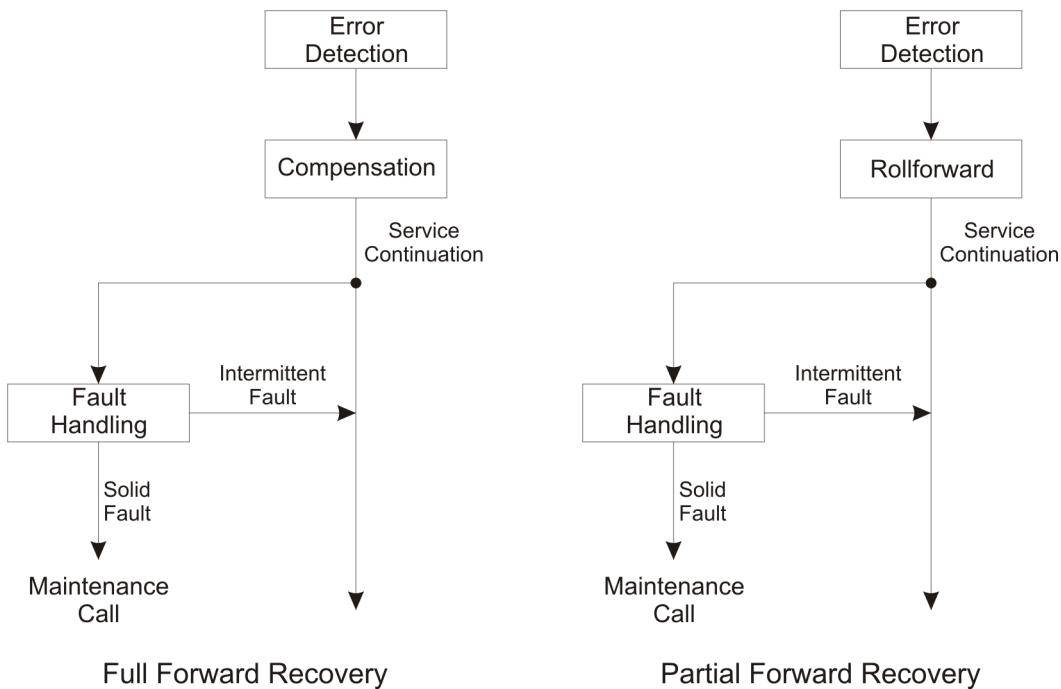


Figura 22 – Estratégia *Forward Recovery* (Avizienis, Laprie et al. 2004)

O sistema de controle de vendas consiste em um agente de banco de dados, um conjunto de pontos de controle e um conjunto de pontos de venda, conforme ilustrado na Figura 23. A principal função deste sistema é manter um banco de dados com a descrição de todos os produtos a serem vendidos de tal forma que os vários pontos de venda distribuídos podem obter os preços corretos dos itens selecionados pelos clientes. Os vários pontos de controle fornecem as interfaces que permitem que os gerentes do sistema atualizem a informação do produto em tempo de execução. Supõe-se que a atualização é uma atividade crítica no sistema. Desta forma, como política de proteção contra fraudes, a atualização só pode ser realizada se dois gerentes, sendo um no nível de gerente sênior, concordarem com a atualização. Logo, é necessário atualizar a informação cooperativamente a partir dos pontos de controle. Estas atualizações precisam ser atômicas do ponto de vista

do ponto de vendas que pode estar realizando consultas ao banco de dados de forma concorrente com a atualização.

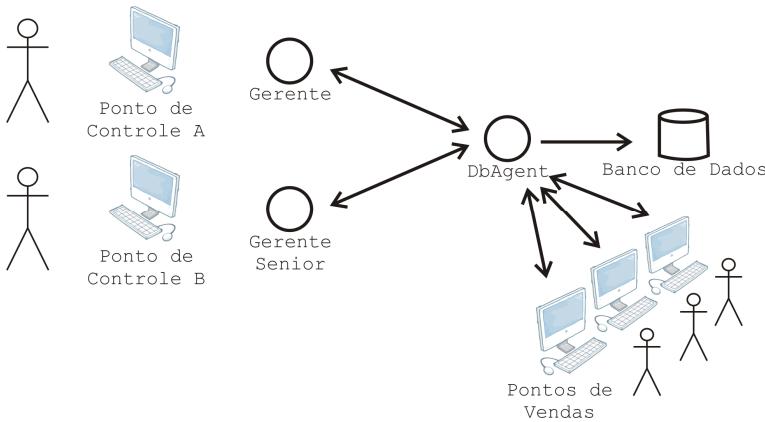


Figura 23 – Sistema de Controle de Vendas

A especificação completa da lei é apresentada no Código 34. Este código é descrito em detalhes no decorrer da discussão dos três cenários abaixo: *REQUISITO 1, SITUAÇÃO 1 e SITUAÇÃO 2*.

REQUISITO 1: ATUALIZAÇÕES PRECISAM SER ATÔMICAS.

A abordagem usual para resolver este problema é através da aplicação da estratégia de *backward recovery* quando houver um problema com a confirmação do segundo gerente. Neste estudo de caso, a estratégia foi implementada através do uso combinado dos elementos de lei: *actions*, *constraints* e o protocolo de interação. O protocolo de interação mostrado na Figura 24 define dois principais caminhos de evolução: as transições {t1, t2} ou {t3, t4}. O caminho {t1, t2} significa que o gerente sênior fez a primeira atualização, enquanto que o segundo caminho significa que o gerente sênior realizou a segunda atualização. Em ambos os casos, quando a primeira transição dispara (t1 ou t3), a *action* *keepContent* é invocada. Esta *action* armazena o conteúdo da atualização no contexto da cena. Desta forma, o conteúdo pode ser utilizado depois por algum outro elemento da lei. De fato, este conteúdo é utilizado pela *constraint* *checkContent*. Esta *constraint* verifica se o conteúdo da segunda atualização é igual ao conteúdo anterior enviado na primeira atualização. Se for igual, então a transição (t2 ou t4) dispara e o agente *dbAgent* atualiza o banco de dados de forma atômica.

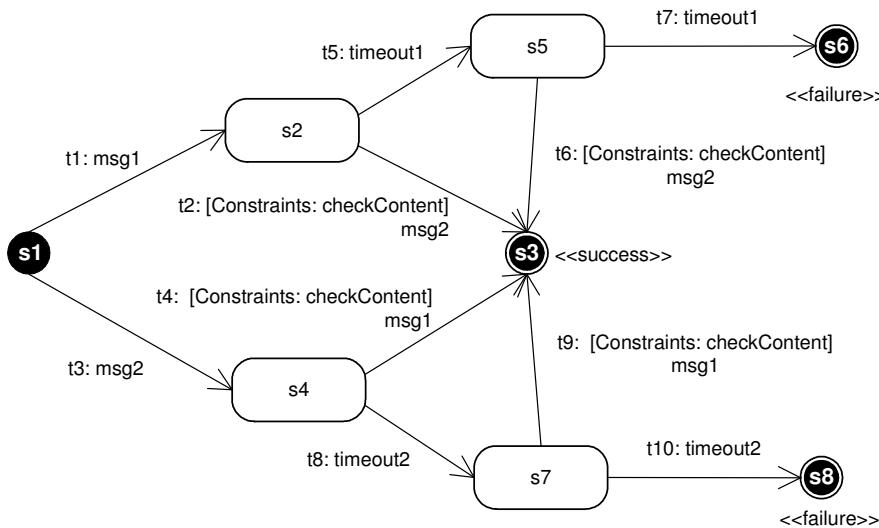


Figura 24 – Protocolo de Interação do Sistema de Controle de Vendas

```

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:}
  
```

Código 34 – Especificação da Lei do Sistema de Controle de Vendas

SITUAÇÃO 1: O SEGUNDO GERENTE NÃO RESPONDE

Como a informação precisa ser atualizada de forma cooperativa, é necessário que o segundo gerente efetue a operação através de uma mensagem de atualização. Neste caso, optou-se por utilizar a estratégia *full forward recovery* para a situação onde não existe uma confirmação de atualização do segundo gerente. Como pode ser visto na Figura 22, existem três atividades principais: detecção do erro, compensação e gerenciamento da falta. A lei especifica como realizar estas atividades. A **detecção do erro** é realizada quando se percebe que o segundo gerente não está respondendo. Os *clocks* nas linhas 18 e 19 são ativados quando a mensagem do primeiro gerente é enviada. Então, os *clocks* contam 2 minutos (120000 milisegundos) que é a quantidade de tempo que o segundo gerente possui para enviar a segunda mensagem de atualização. Se o segundo gerente não enviar a mensagem dentro deste período, o clock gera um evento de *clock_tick*.

Capturar este evento significa perceber que o erro ocorreu, neste caso, o gerente não está respondendo. Após a detecção do erro, pode-se realizar uma estratégia de **compensação**. No estudo de caso, esta estratégia é relativamente simples e consiste em enviar uma mensagem de *broadcast* para todos os agentes alertando que existe uma atualização pendente por causa da falta de confirmação de um segundo gerente. Na lei, isto é feito através da action *warnManagerBroadcast* na linha 24. Esta action é ativada somente quando as transições t5 ou t8 são disparadas em consequência do evento *clock_tick*.

O *clock* é declarado como periódico. Isto significa que ele permanece ciclicamente gerando eventos a cada dois minutos até ele se tornar inativo através das transições t2, t6, t4 ou t9 (linhas 18 a 19). Portanto, os gerentes tem mais dois minutos para tomar uma atitude em resposta à mensagem de *broadcast warnManagerBroadcast*. Se algum gerente responde a esta mensagem com uma confirmação de atualização, então as transições t6 ou t9 são disparadas e o protocolo finaliza com sucesso. Caso contrário, se não houver nenhuma resposta do gerente, é preciso realizar o **gerenciamento de faltas**. Este caso é gerenciado pela action *handleTimeout* na linha 22. Esta action envia uma mensagem a todos os agentes envolvidos na conversação informando que o segundo gerente não respondeu e, portanto, cada agente deve realizar a sua própria estratégia de recuperação.

Embora muitas complexidades tenham sido omitidas em prol da objetividade e simplicidade, o exemplo é suficientemente detalhado para ilustrar como as leis podem incorporar preocupações de fidedignidade. Mais especificamente, neste estudo de caso, isto foi realizado através da especificação de uma estratégia *full forward recovery* através da detecção do erro, compensação e gerenciamento de faltas.

SITUAÇÃO 2: GERENTES ENVIAM CONTEÚDOS DE ATUALIZAÇÃO DIFERENTES

Para que haja a confirmação da mensagem de atualização do primeiro gerente, o segundo gerente precisa enviar outra mensagem com exatamente o mesmo conteúdo da primeira mensagem. Para lidar com esta situação, propõe-se a utilização de uma estratégia de tolerância a faltas. Primeiramente, é feita a detecção do erro através do uso de *constraints* e *actions*. Depois disso, o gerenciamento da falta é realizado para fazer com que os agentes envolvidos na conversação estejam cientes da falha. Em relação a **deteção do erro**, a *action* *keepContet* armazena o conteúdo da primeira mensagem e a *constraint* *checkContent* verifica se o conteúdo da segunda mensagem é igual ao conteúdo da primeira mensagem. Se a *constraint* *checkContent* descobre que os conteúdos não são iguais, então gera-se o evento *constraint_not_satisfied*. Este evento é capturado pela *action* *handleDifferentContet*. Esta *action* executa o **gerenciamento de faltas** informando todos os agentes participantes que existe um conteúdo inesperado. Isto dá aos gerentes outra oportunidade para enviar a mensagem correta.

De fato, esta estratégia realiza um **partial forward recovery**. Ela detecta erro, mantém o sistema em um estado seguro (uma vez que nem a transição t2 nem t4 são disparadas porque as *constraints* não permitem), realiza o gerenciamento da falta e, no caso de o segundo gerente enviar outra mensagem com o conteúdo esperado, o protocolo finaliza com sucesso.

6.1.3.Trabalhos Relacionados

Do ponto de vista de fidedignidade, a abordagem LGI (Minsky and Ungureanu 2000) possui ênfase principal em segurança (*security*) e confiança (*trust*). A arquitetura prevê a utilização de entidades certificadoras, criptografia e

um conjunto de operações em seu modelo conceitual para este fim. Entretanto, não foi observado que o LGI tenha explicitamente incorporado preocupações de fidedignidade ou gerenciamento de faltas.

O segundo trabalho relacionado a esta tese são as Instituições Eletrônicas. Entretanto, embora elas possuam uma forte correlação com o modelo conceitual do XMLaw, em nenhum dos artigos conhecidos foi possível observar preocupações explícitas com fidedignidade (Rodriguez-Aguilar 2001; Dignum 2002; Esteva 2003; Esteva, Rosell et al. 2004; Garcia-Camino, Noriega et al. 2005; Ashri, Payne et al. 2006; Weyns, Omicini et al. 2007).

6.1.4.Considerações Finais

O objetivo deste capítulo foi discutir o relacionamento entre as leis de interação e os conceitos relacionados a fidedignidade. Mostrou-se como implementar estratégias de tolerância a faltas utilizando-se o XMLaw. Ao usar as leis para especificar preocupações de fidedignidade, permite-se a reutilização de toda a infra-estrutura de monitoramento e *enforcement* disponível nas abordagens de leis. Além disso, a fidedignidade é definida explicitamente e de forma predominantemente declarativa. O modelo de eventos do XMLaw contribuiu para compor elementos tais como transições, normas e relógios. Esta facilidade de composição é um dos fatores que permitiram a incorporação de preocupações de fidedignidade nas leis.

6.2. Dependability Explicit Computing e Leis

Muitos dos sistemas atuais são abertos e dinâmicos. Uma característica chave é que eles demandam algum tipo de ligação (*binding*) dinâmica, ou seja, a seleção e o uso de componentes ou agentes em tempo de execução. Portanto, não se trata apenas de agentes selecionados durante a atividade de projeto do sistema, mas ao invés disso os sistemas são abertos para permitir a chegada, partida ou modificação dos agentes. Uma das formas de se promover fidedignidade neste tipo de sistema é através da abordagem intitulada *Dependability Explicit Computing* (DepEx) (Kaâniche, Laprie et al. 2000). DepEx trata os metadados de fidedignidade como informações de primeira classe. Os meios para fidedignidade (prevenção de faltas, tolerância a faltas, previsão de faltas e remoção de faltas)

devem ser explicitamente incorporados em um modelo de desenvolvimento focado em sistemas fidedignos.

Durante o desenvolvimento do sistema, os artefatos são especificados através da incorporação de informações relacionadas a fidedignidade desde as fases iniciais do desenvolvimento do sistema. As informações são então atualizadas a medida que o desenvolvimento evolui.

De posse das informações de fidedignidade, é possível utilizá-las para auxiliar na tomada de decisão tanto em tempo de execução quanto em tempo de projeto. Por exemplo, um desenvolvedor pode escolher o componente que utilizará baseado em seu tempo de resposta médio obtido durante o histórico de execução do sistema. Outro desenvolvedor pode escolher um componente baseado nos algoritmos de criptografia que o componente disponibiliza. Alguns exemplos de metadados são taxa de falhas, modos de falhas, pré e pós-condições, MTBF (*mean time between failures*), confiabilidade, tempo de resposta, recursos consumidos, especificação do componente, faltas conhecidas, tipos de criptografia, etc.

Neste capítulo, propõe-se a incorporação das idéias de DepEx na abordagem de leis XMLaw. Mostra-se que as leis podem explicitamente coletar dados relacionados a fidedignidade e publicá-los em um banco de dados de fidedignidade. As informações deste banco de dados podem ser utilizadas para tornar concretas as idéias de DepEx e, por exemplo, guiar decisões de projeto ou em tempo de execução. As principais vantagens para a utilização de uma abordagem de leis são: (i) a definição explícita de preocupações de fidedignidade; (ii) a coleta automática de metadados de fidedignidade reutilizando a infra-estrutura do mediador presente no M-Law; e (iii) a habilidade de especificar reações em resposta a situações indesejáveis prevenindo a ocorrência de falhas no sistema.

6.2.1.Implementação de DepEx usando XMLaw

Nesta seção, apresenta-se um estudo de caso para ilustrar como especificar as leis de tal forma que metadados de fidedignidade sejam tratados como entidades de primeira classe. Este problema foi publicado em (Yi and Kochut 2004) e foi ligeiramente modificado para este estudo de caso.

6.2.1.1.Descrição do Problema

Considere a tarefa de criar um sistema composto de três tipos de agentes: um agente de viagens, um agente do usuário e uma agente de uma companhia aérea. A companhia aérea fornece uma série de operações relacionadas que precisam ser chamadas de acordo com um complexo protocolo de interação. As operações fornecidas são *checkSeatAvailability*, *reserveSeats*, *cancelReservation*, *bookSeats* e *notifyExpiration*. Estas operações precisam ser invocadas pelos clientes de acordo com as seguintes regras de conversação:

- *checkSeatsAvailability*: precisa ser a primeira operação a ser chamada;
- *reserveSeats*: só pode ser chamada se o cliente invocou anteriormente a operação *checkSeatsAvailability* e os assentos requisitados estavam disponíveis. A reserva só é guardada por um determinado período de tempo;
- *bookSeats* ou *cancelReservation*: podem ser invocados somente se os assentos foram previamente reservados (através da invocação com sucesso da operação *reserveSeats*) e a reserva não tiver expirado.;
- se nem a operação *bookSeats* nem *cancelReservation* for invocada pelo cliente dentro do tempo especificado, o agente da companhia aérea invocará a operação *notifyExpiration* para informar ao cliente que a reserva expirou.

O fluxo básico de interação descrito a seguir é ilustrado no diagrama de seqüência da Figura 25. O viajante, representado pelo agente do usuário, que planeja realizar uma viagem, submete uma ordem de compra de viagem (*TripOrder*) através da mensagem *getItinerary* para o agente de viagens. O agente de viagem deve responder a esta mensagem com uma proposta de itinerário (*Itinerary*). A ordem de compra de viagem submetida pelo agente usuário contém informações tais como partida, destino, horário e dia de partida e destino, número máximo de conexões e número de viajantes. O agente de viagens procura pelo melhor itinerário para satisfazer as exigências do viajante, considerando também os critérios tais como menor preço, disponibilidade dos vôos e milhas acumuladas pelo viajante. Antes do itinerário ser proposto ao viajante, o agente de viagens se comunica com o agente da companhia aérea para verificar a disponibilidade dos assentos (*checkSeatsAvailability*). Se não houver assentos disponíveis, o agente de viagens notifica o agente do usuário e espera que o usuário emita uma *TripOrder*.

modificada. Se, por outro lado, houver assentos disponíveis, o itinerário proposto é enviado para o agente do usuário para a confirmação. O agente do usuário decide então reservar os assentos para o itinerário proposto e informa ao agente de viagens mais informações pessoais para que o agente da companhia aérea possa lhe enviar diretamente o bilhete eletrônico da sua passagem aérea.

No passo seguinte, o agente de viagens interage com o agente da companhia aérea para finalizar a reserva (*reserveSeats*). A companhia aérea garante a reserva por um prazo de um dia e se ela não receber uma mensagem *BookRequest* neste período, os assentos são liberados e o agente de viagens é notificado. O agente de viagens envia uma mensagem *ReserveResult* para o agente do usuário para informar a resposta da reserva.

Neste ponto, o viajante pode tanto confirmar quanto cancelar a reserva. Se ele decide confirmar, então ele envia a mensagem *BookRequest* para o agente de viagens contendo a informação do cartão de crédito. Finalmente, o agente de viagens invoca a operação *bookSeats* do agente da companhia aérea. Como resultado, o agente da companhia aérea garante os assentos para o itinerário proposto e emite um bilhete eletrônico para o agente do usuário.

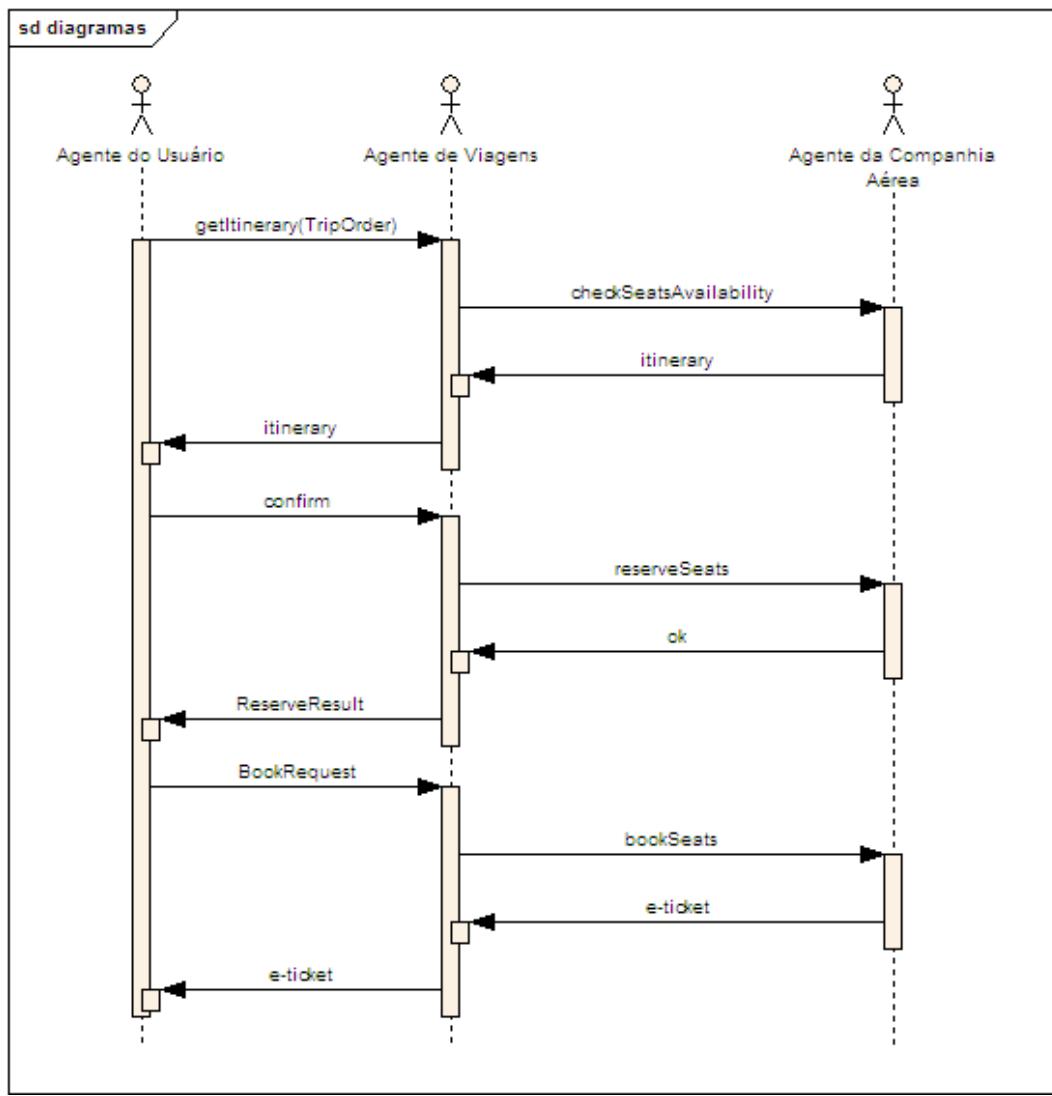


Figura 25 – Diagrama de Seqüência do Estudo de Caso

6.2.1.2. Arquitetura

A arquitetura do sistema é mostrada na Figura 26. Esta arquitetura é baseada no modelo arquitetural para tratar metadados de fidedignidade proposto em (Serugendo, Fitzgerald et al. 2006). Esta arquitetura foi concebida para alcançar níveis previsíveis de recuperação em casos de falhas em sistemas distribuídos. Neste estudo de caso, escolheu-se esta arquitetura porque ela já contém os componentes necessários para habilitar os conceitos de DepEx. A arquitetura prevê um banco de dados de metadados, um serviço de adaptação/inferência em tempo de execução e um componente para aquisição dos metadados.

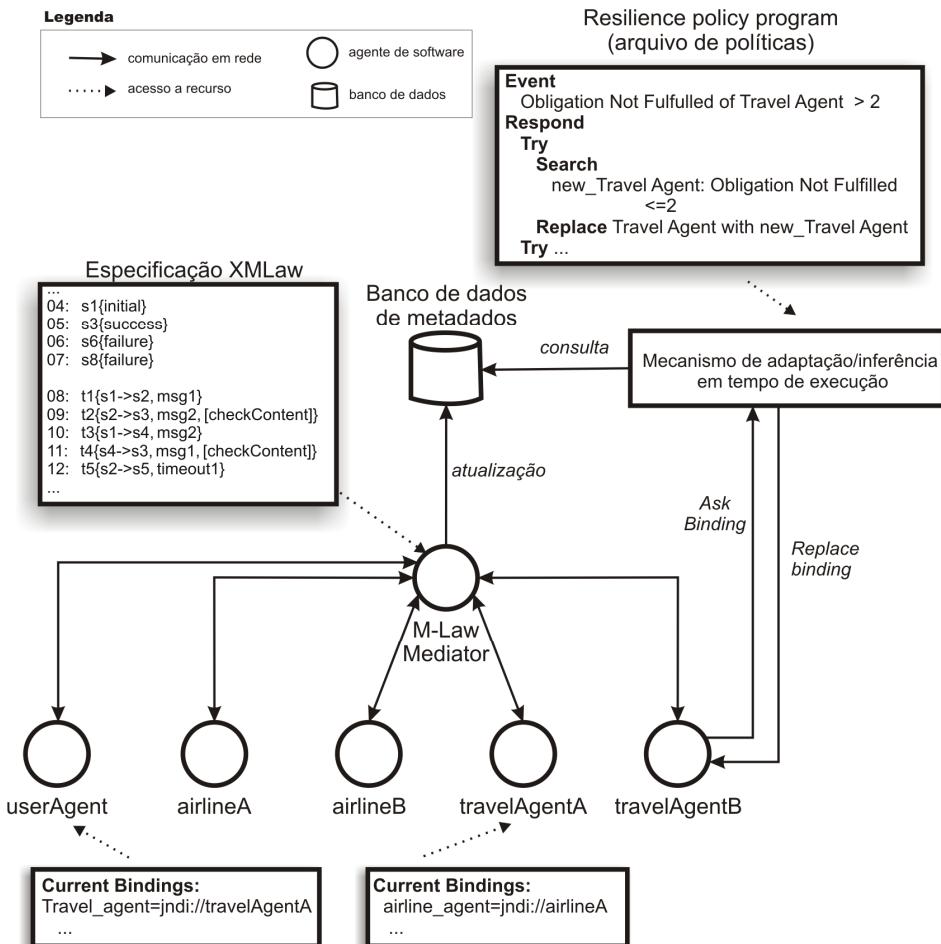


Figura 26 – Arquitetura do Estudo de Caso de DepEx

Neste estudo de caso, o papel do componente de aquisição de metadados é desempenhado pelo M-Law. O M-Law funciona mediando a comunicação entre os agentes. O comportamento do mediador é especificado no arquivo de lei XMLLaw que ele lê. No contexto da aquisição de dados, a especificação XMLLaw conterá instruções que dizem ao mediador como atualizar o banco de dados de metadados. Em tempo de execução, este banco de dados pode ser utilizado de duas maneiras: pelos próprios agentes ou pelo mecanismo de adaptação dinâmica. Os agentes podem, proativamente, realizar consultas no banco de dados e se adaptarem para refletir as exigências de seus requisitos de fidedignidade. O agente do usuário do exemplo anterior pode buscar informações sobre os agentes de viagens que não tiverem descumprido nenhuma obrigação durante o último mês.

Neste estudo de caso, existem dois agentes de viagens disponíveis: *travelAgentA* e *travelAgentB*. Em tempo de execução, o agente de usuário é capaz de escolher com qual deles irá interagir baseado nas informações de fidedignidade presentes no banco de dados. Por sua vez, os agentes de viagens possuem dois

agentes de companhias aéreas com os quais eles podem interagir. A escolha de qual deles é utilizado também pode ser baseada nos atributos de fidedignidade disponíveis no banco de dados.

O mecanismo de adaptação dinâmica fornecido na arquitetura fornece serviços relacionados ao processamento dos metadados armazenados no banco de dados. Exemplos destes serviços são a comparação de metadados, determinação de informações equivalentes e composição de metadados (Serugendo, Fitzgerald et al. 2006). Este serviço também pode ser utilizado para conectar os agentes de acordo com a especificação do arquivo de políticas (ilustrado através do *Resilience Policy Program* na Figura 26). Entretanto, o foco deste estudo de caso é ilustrar como se pode utilizar as leis para atualizar automaticamente o banco de dados de metadados.

6.2.1.3.Os Metadados

As leis irão especificar metadados relacionados a disponibilidade, falha de serviço e *enforcement* de pré e pós condições.

- Disponibilidade – todas as vezes que um agente envia uma requisição para outro agente, o destinatário deve responder dentro de um período pré-definido de tempo. A ausência de uma resposta implica que naquele momento o destinatário não estava disponível com o nível de qualidade desejado (indicado pela quantidade de tempo).
- Falha de serviço – durante a interação, os agentes adquirem obrigações que eles precisam cumprir. O cumprimento destas obrigações representa o comportamento correto esperado para os agentes. Portanto, toda vez que uma obrigação é descumprida, ela pode ser interpretada como uma falha de serviço, ou seja, a execução do sistema difere do comportamento esperado.
- Pré e pós condições – As especificações dos agentes podem mudar a medida que os agentes evoluem. A especificação de quais são os serviços fornecidos e as pré e pós condições para estes serviços são importantes para que seja possível utilizar mecanismos de adaptação dinâmica e realizar, por exemplo, a composição dinâmica do sistema. Como um exemplo de pré-condição, suponha-se que se deseja garantir que os valores dos atributos *departure* e *destination* da ordem de compra (*TripOrder*) da mensagem *getItinerary* sejam valores pertencentes ao conjunto $S=\{\text{"Toronto"}, \text{"New York"}, \text{"London"}, \text{"Tokyo"}, \text{"Rio de Janeiro"}\}$. Garantir o cumprimento desta restrição implica que o agente de viagens receberá um parâmetro que está dentro do escopo da especificação do sistema.

Considerando-se a descrição do estudo de caso apresentado na Seção 6.2.1.1 e os metadados descritos acima, é possível especificar os seguintes requisitos para a especificação das leis:

Requisito #1 – Todo o processo de interação precisa ocorrer dentro de dois dias. Após dois dias o processo é cancelado e todas as regras se tornam inválidas. Todas as interações em andamento precisam ser reiniciadas.

Requisito #2 – Todas as interações precisam ocorrer na ordem pré-definida de acordo com o especificado na descrição do problema na Seção 6.2.1.1.

Requisito #3 – Se o agente da companhia aérea informa que existe um assento disponível, o assento deve ser reservado para o agente de viagens por pelo menos cinco minutos. Desta forma, o usuário possui algum tempo para decidir sobre a sua confirmação de reserva. Se os cinco minutos passarem e a companhia aérea não receber nenhuma confirmação, então é permitido que a companhia aérea responda com uma mensagem *not-available* a possíveis tentativas de confirmação. O assento pode ser reservado para outro cliente.

Requisito #4 – Quando o agente da companhia aérea envia a mensagem *result-ok* em resposta a uma reserva de assento, a reserva precisa ser guardada por pelo menos um dia.

Requisito #5 – O conteúdo de *TripOrder* precisa pertencer o conjunto de possíveis valores $S=\{\text{"Toronto"}, \text{"New York"}, \text{"London"}, \text{"Tokyo"}, \text{"Rio de Janeiro"}\}$

Requisito #6 – Cada requisição que não precisar de interação com o usuário precisa ser respondida dentro de 15 segundos por qualquer agente.

6.2.1.4. Aquisição de Metadados Através da Especificação de uma Lei em XMLaw

O protocolo de interação é mostrado na Figura 27 e a especificação completa da lei pode ser vista no Código 35. A cena é declarada nas linhas 01 e 02. As linhas 03 a 16 contêm padrões de mensagens que se espera que os agentes troquem entre si. As linhas 17 a 20 especificam os estados iniciais e finais do protocolo de interação. As transições são especificadas nas linhas 21 a 37. As transições referenciam estados, mensagens, *constraints* e normas presentes na lei. Os *clocks* são especificados nas linhas 38 a 40, a *constraint* na linha 41, *actions*

nas linhas 42 a 44 e as normas nas linhas 45 e 46. A seguir, mostra-se como os seis requisitos de leis foram utilizados para especificar as leis.

Requisito #1: este requisito é implementado através do atributo *time-to-live* da cena *planningATrip* (linha 02).

Requisito #2: o protocolo de interação da Figura 27 reflete exatamente os possíveis caminhos de interação descritos neste estudo de caso. Este protocolo é especificado nas linhas 03 a 37. Estas linhas declaram mensagens, estados e transições presentes no protocolo.

Requisito #3: este requisito demanda pelo uso combinado de vários dos elementos do XMLaw. Primeiramente, é necessário identificar quando o agente da companhia aérea “informa que existe um assento disponível”. Depois, é preciso contar cinco minutos. Não é permitido que a companhia aérea responda com a mensagem *not-available* nestes cinco minutos. A Tabela 10 mostra como a observação da seqüência de eventos torna possível especificar este requisito. Esta tabela é mapeada para o XMLaw nas linhas 35, 39, 43 e 45.

<p>O agente da companhia aérea envia a mensagem <i>itinerary-1</i> para o agente de viagens. Isto significa que a companhia aérea está dizendo “existe um assento disponível”. Então, ativa-se um <i>clock</i> para contra o tempo. Além disso, também ativa-se a obrigação <i>hold-seat</i> para o agente da companhia aérea.</p>	
WHEN (t3, transition_activation)	ACTIVATE hold-seat-clock, hold-seat
Se o prazo que a companhia aérea possui para aguardar o assento expirar (evento <i>clock_tick</i>), então a obrigação <i>hold-seat</i> não precisa mais ser cumprida, ou seja, o agente da companhia aérea pode responder com uma mensagem <i>not-available</i> .	
WHEN (hold-seat-clock, clock_tick)	DEACTIVATE hold-seat
Se a companhia aérea responde com um <i>result-ok</i> a uma requisição <i>reserveSeats</i> , isto significa que a companhia aérea cumpriu a sua obrigação de reservar o assento. Logo, a obrigação deve ser desativada.	
WHEN (t7, transition_activation)	DEACTIVATE hold-seat
A transição <i>t15</i> só dispara se a obrigação <i>hold-seat</i> estiver desativada. Se a agência de viagens envia uma mensagem <i>not-available</i> enquanto a obrigação ainda estiver ativa, então o evento <i>norm_not_fulfilled</i> irá ser gerado e a transição não irá disparar. Como neste estudo de caso, o foco é a utilização das leis para a aquisição de metadados, o evento de não cumprimento de uma obrigação deve ser reportado para o banco de dados de metadados. A <i>action updateHoldSeatMetadata</i> é responsável pelas informações do contexto tais como a identificação do agente e da obrigação (neste caso <i>hold-seat</i>) e atualizar o banco de dados.	
WHEN (hold-seat, norm_not_fulfilled)	ACTIVATE updateHoldSeatMetadata

Tabela 10 – Linha de Raciocínio para a Especificação XMLaw do Requisito #3 nas Linhas 35, 39, 43 e 45.

Requisito #4: este requisito é especificado no XMLaw utilizando-se uma idéia similar ao requisito #3. A transição *t16* (linha 36) somente dispara se a obrigação *hold-reservation* (linha 46) estiver desativada. O *clock hold-reservation-clock* (linha 40) conta o tempo até um dia. A *action updateHoldReservationMetadata* atualiza o banco de dados de metadados com a informações sobre os agentes que não cumpriram esta obrigação.

Requisito #5: a *constraint checkContent* especificada na linha 41 é invocada pela transição *t1* (linha 21). Esta *constraint* verifica se os valores das variáveis *dep* e *dest* (linha 03) pertencem ao conjunto pré-definido de cidades. A implementação da *constraint* é apresentada na listagem de Código 36.

Requisito #6: este requisito indica que os agentes que não precisam esperar por uma resposta do usuário não demorem para responder a mensagens. O *clock availability-clock* especificado na linha 38 conta 15 segundos todas as vezes que um agente recebe uma mensagem. O *clock* é reiniciado quando o agente responde às mensagens. As transições *t1,t2,t3,t5,t6,t7,t9,t10,t11*, e *t13* especificada no *clock* representam as requisições para os agentes. Nota-se que transições tais como *t4* não estão presentes nesta lista. Isto é porque *t4* representa uma mensagem que é enviada para o usuário (através do agente do usuário). Todas as vezes que um agente não responder à requisição dentro dos 15 segundos, o *clock* gera um evento *clock_tick*. Este evento é observado pela *action updateClockMetadata* (linha 42). Esta *action* atualiza o banco de dados de metadados indicando que o agente não estava disponível naquele momento. O implementação desta *action* é mostrado no Código 37.

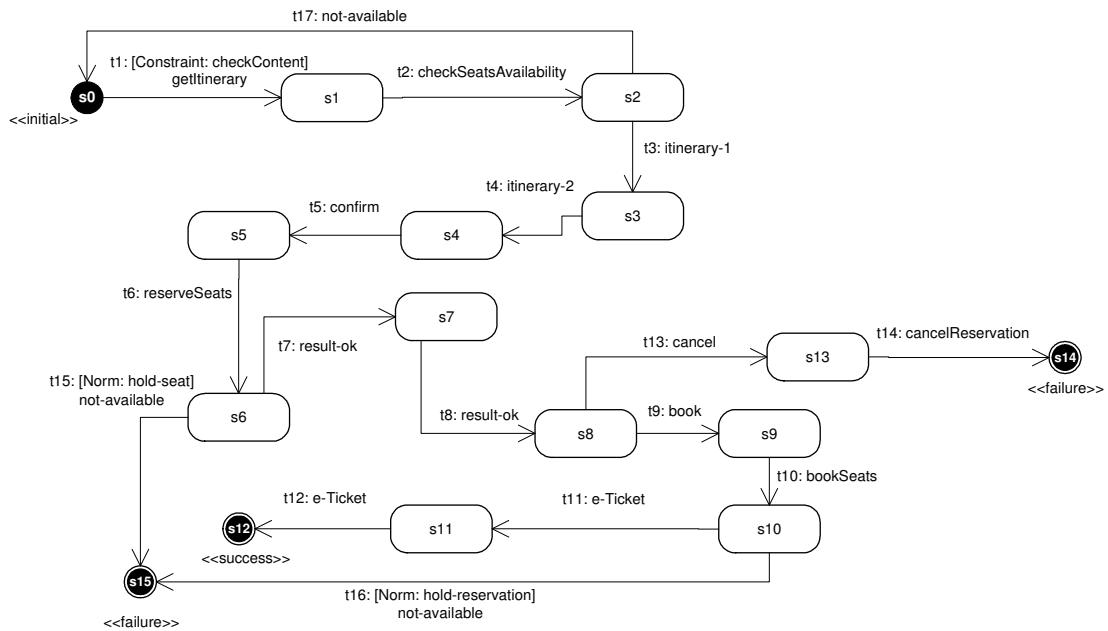


Figura 27 – Protocolo de Interação do Estudo de Caso de DepEx

```
// 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: }
```

Código 35 – XMLaw do Estudo de Caso de DepEx

```

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
        }
    }
}

```

Código 36 – Implementação Java da constraint *checkContent*

```

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);
    }
}

```

Código 37 – Action *updateClockMetadata* implementada através da classe Java *DecAvailability*

6.2.1.5.O Banco de Dados de Metadados

Neste estudo de caso, a estrutura do banco de dados é composta por duas entidades: *agent* e *dependability_data*. O modelo entidade-relacionamento é mostrado na Figura 28 e os seus atributos são descritos na Tabela 11.

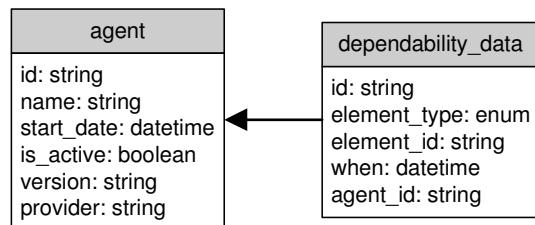


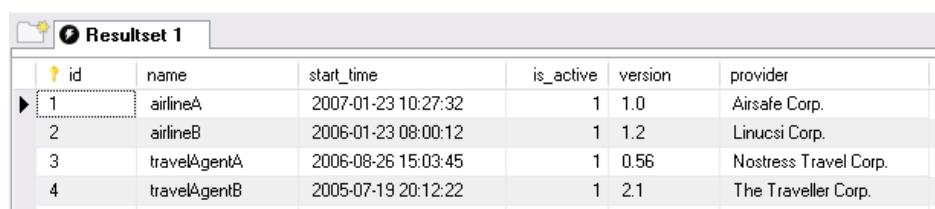
Figura 28 – Modelo Entidade-Relacionamento do Banco de Dados de Metadados

agent	dependability_data
id – identificador único do agente no banco de dados	id – identificador único da tupla no banco de dados
name – nome do agente. O nome precisa ser único	element_type – tipo de elemento XMLaw. (ex: obligation, clock, ...)
start_date – data e hora de quando o agente foi adicionado ao banco de dados	element_id – identificador do elemento XMLaw
is_active – possui o valor true se o agente está em execução	when – data e hora do registro desta tupla no banco de dados
version – versão do agente	agent_id – identificação do agente associado com esta tupla
provider – organização responsável pela criação do agente	

Tabela 11 – Descrição dos Atributos do Banco de Dados

As *actions* *updateClockMetadata*, *updateHoldSeatMetadata*, *updateHoldReservationMetadata* (linhas 42, 43 e 44) são responsáveis por atualizar o banco de dados de metadados. De fato, estas *actions* atualizam as informações de fidedignidade dos agentes em tempo de execução. A Figura 29 e Figura 30 mostram telas capturadas do banco de dados de metadados. Por exemplo, na Figura 29 mostra-se que o agente *airlineA* (*agent_id*=1) não cumpriu a obrigação *hold-seat* no dia primeiro de Fevereiro e a obrigação *hold-reservation* no dia 3 de Fevereiro.

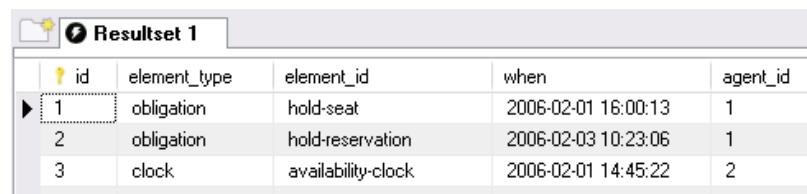
É importante perceber que embora as *actions* sejam elementos relativamente simples que obtêm informações do contexto e atualizam o banco de dados, é a especificação das leis que diz quando as *actions* devem executar. Em outras palavras, a aquisição das informações de fidedignidade é feita através do uso combinado de vários dos elementos do XMLaw.



The screenshot shows a database result set titled "Resultset 1". It contains a table 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.

Figura 29 – Exemplos de Agentes Cadastrados no Banco de Dados



The screenshot shows a database result set titled "Resultset 1". It contains a table 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

Figura 30 – Exemplos de Informações de Fidedignidade

O banco de dados de metadados pode ser utilizado para a realização de diversos tipos de consultas. Por exemplo, para consultar o número de obrigações que não foram cumpridas pelo agente *airlineA*, pode-se escrever o seguinte comando SQL:

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

6.2.2.Trabalhos Relacionados

Em (Chen, Li et al. 2005), foi apresentada uma ferramenta para monitorar a fidedignidade e o desempenho de *Web Services*. A ferramenta coletava os metadados agindo como cliente dos Web Services sob investigação. Os resultados são coletados e disponibilizados em um banco de dados disponível publicamente. A ferramenta monitora um dado Web Services em relação a um conjunto fixo de características:

- (i) disponibilidade: periodicamente a ferramenta realizava requisições *dummy* para os Web Services para verificar se eles estavam em execução;
- (ii) funcionalidade: a ferramenta realiza chamada para os Web Services e verifica os resultados retornados;
- (iii) desempenho: a ferramenta monitora o tempo entre a requisição e o recebimento de uma chamada, produzindo estatísticas em tempo de execução sobre o desempenho de um determinado Web Service;
- (iv) falhas e exceções: a ferramenta registra falhas e exceções ocorridas durante o período de teste do Web Service para análise futura.

Embora a ferramenta seja útil para muitas aplicações existentes, quando comparada com a solução apresentada neste capítulo, as leis fornecem uma maneira muito mais expressiva e flexível de coletar informações específicas de uma determinada aplicação. Por exemplo, através da ferramenta não seria possível expressar nenhuma das obrigações utilizadas no estudo de caso.

Não foi encontrado na literatura uma solução relacionada que englobasse as várias características encontradas na proposta deste capítulo:

- (i) *enforcement* do comportamento de interação;
- (ii) especificação flexível (principalmente por causa do modelo de eventos) e declarativa das interações;
- (iii) incorporação de preocupações de fidedignidade na especificação;
- (iv) um banco de dados publicamente disponível com informações de fidedignidade.

6.2.3.Considerações Finais

Neste capítulo, mostrou-se que DepEx e leis são abordagens complementares e integráveis. As leis podem ser ferramentas adequadas para monitorar e especificar metadados de fidedignidade relativamente complexos. Mais precisamente, apresentou-se a incorporação de DepEx na abordagem de XMLaw. Um estudo de caso detalhado também foi apresentado, ressaltando principalmente o papel de aquisição de informações de fidedignidade. O estudo de caso apresentado possui três contribuições principais:

- (i) mostra a integração de um mecanismo de lei (M-Law) em uma arquitetura de adaptação dinâmica baseada em dados de fidedignidade;
- (ii) ilustração de que as leis podem ser não somente uma maneira adequada de coletar informações de fidedignidade, mas também de interferir na execução do sistema quando necessário;
- (iii) com as leis as preocupações de fidedignidade são explicitamente consideradas e precisamente especificadas de forma predominantemente declarativa.

Também foi apresentado um modelo entidade-relacionamento de um banco de dados de fidedignidade e como este modelo pode ser consultado para retornar informações relevantes.

A abordagem utilizada neste capítulo possui as vantagens de flexibilidade e reutilização. Flexibilidade porque em contraste com os trabalhos relacionados, o alto nível de abstrações presentes no XMLaw permitem uma maneira expressiva de capturar metadados específicos do domínio. Reutilização porque não é

necessário construir uma nova linguagem nem um novo mediador para realizar a aquisição de metadados.

7**Estudo de Caso: Controle de Tráfego Aéreo**

A principal função do controle de tráfego aéreo (CTA) é monitorar e regular o tráfego aéreo de um aeroporto de origem até um aeroporto de destino (Ndovie 1994). Este tipo de sistema é frequentemente caracterizado como de larga escala, com alta complexidade e dinamismo. Tipicamente, sistemas desta natureza precisam controlar e monitorar milhares de aeronaves em um cenário onde as condições do ambiente estão em constante mudança e diante de situações imprevistas. Os dois principais atores de um CTA são o piloto e o controlador. Os controladores precisam lidar com situações muitas vezes complexas em intervalos de tempo bastante rígidos. Além disso, tanto as ações dos pilotos quanto dos controladores precisam estar em conformidade com as regras definidas pelas agências reguladoras. No Brasil, a agência responsável pelo CTA é a Agência Nacional de Aviação Civil (ANAC).

Como consequência deste cenário onde decisões complexas precisam ser tomadas com restrições de tempo bastante rígidas, os pilotos e controladores são claramente um ponto de risco no sistema e cuja falha pode acarretar consequências desastrosas. O grande número de variáveis do domínio de CTA aumenta a probabilidade de falhas. A ocorrência de falhas pode ser desastrosa dada a criticidade do sistema. Torna-se, portanto, necessário a aplicação de técnicas que auxiliem a construção de sistemas de CTA fidedignos.

O principal objetivo deste estudo de caso é ilustrar como a abordagem de governança proposta nesta tese pode ser aplicada em um domínio complexo para melhorar na sua fidedignidade.

7.1.Leis em CTAs

Alguns dos empecilhos na melhoria da fidedignidade dos sistemas de CTA atuais residem na dificuldade de verificar se as aplicações comerciais “*off-the-shelf*” se comportam de acordo com o esperado e na dificuldade em traduzir o conhecimento técnico do domínio de CTA para o projeto do software (Matthews

2002). A abordagem de governança proposta nesta tese auxilia na diminuição destas dificuldades da seguinte forma:

- “...dificuldade de verificar se as aplicações comerciais *off-the-shelf...*” – A estrutura de monitoramento implementada no middleware M-Law assume que os agentes são tratados como caixas-pretas. Ou seja, não é feita nenhuma suposição sobre os detalhes internos de implementação ou arquitetura dos agentes. Assume-se apenas que os agentes se comunicam através de troca de mensagens mediadas pelo M-Law. Desta forma, as aplicações comerciais *off-the-shelf* podem ser vistas como agentes de software e, portanto, teriam o seu comportamento verificado de acordo com a especificação das leis.
- “... dificuldade em traduzir o conhecimento técnico do domínio de CTA para o projeto do software ... ” – Conforme pode ser visto no modelo conceitual e na comparação com as abordagens de governança relacionadas, o XMLaw possui abstrações que alto nível que são usadas para a especificação das leis. Estas abstrações tratam de conceitos que estão bem mais próximos do mundo real para especificação do comportamento esperado. Conceitos como cena, obrigação, proibição, permissão e mensagem permitem que os especialistas no domínio de CTA foquem na especificação das regras enquanto os especialistas em TI (Tecnologia da Informação) se preocupem em construir o sistema em conformidade com as especificações de leis e com os seus requisitos funcionais e não-funcionais.

7.2.Fontes de Problemas em Sistemas de CTA

De forma geral os problemas que ocorrem em um sistema de CTA e que acarretam em acidentes podem ser causados pelas seguintes fontes de problemas:

- Falta de conhecimento técnico – a ignorância de fatores técnicos tais como estruturas, materiais e aerodinâmica foi responsável por um grande número de acidentes (Matthews 2002). Entretanto, conforme o conhecimento científico e tecnológico avança, acidentes com esta causa são cada vez mais raros.

- Software – faltas que não tratadas adequadamente induzem a falhas no sistema. Embora não tenha sido encontrado nenhum artigo sobre percentual de falhas de software no total das causas dos acidentes aéreos, o artigo de Rahman et al. (Rahman, Beznosov et al. 2006) mostra que no período de 12 anos entre 1994 e 2005, 36% das falhas apresentadas em infraestruturas consideradas críticas, dentre elas, transporte aéreo, fornecimento de água e transportes ferroviários, foram causadas por software. O segundo maior percentual foram falhas de hardware com 21%, seguido por falhas humanas com 7%.
- Hardware – em sistemas de CTA exemplos de falhas de hardware podem ser um radar que deixou de funcionar, ou o link da comunicação entre o piloto e o controlador que não conseguiu ser estabelecido.
- Problemas durante o vôo – esta categoria de problemas agrupa as situações como terrorismo, passageiros sob efeito de álcool e brigas, dentre outras.
- Falha Humana – com o avanço do conhecimento técnico, falhas nos aviões tem se tornado cada vez mais raras. Isto tem levado a exposição das falhas humanas. Em 2004, nos Estados Unidos, a falha de pilotos foi considerada como a principal causa de 78,6% dos acidentes fatais e 75,5% de todos os acidentes ocorridos na aviação civil (Krey 2006).
- Condições do tempo – geralmente esta categoria é classificada como uma subcategoria de falha humana. Embora os acidentes causados por condições adversas de tempo, como temporais, ocorram com freqüência relativamente baixa (4,5% do total de acidentes por falha humana), eles correspondem por 19,7% dos acidentes fatais causados por falha humana (Krey 2006).

Devido à importância das categorias de software e falha humana, optou-se por abordar predominantemente estas duas categorias neste estudo de caso. Para isto, analisou-se o domínio de ATC tanto sob a perspectiva tradicional de governança quanto sob a perspectiva de fidedignidade.

7.3.Engenharia do Sistema

O sistema de controle de tráfego aéreo será utilizado por dois tipos de usuários: controladores e pilotos. A principal função dos controladores é manter uma separação lateral, vertical e longitudinal entre as aeronaves. Além disso, os controladores também devem manter um fluxo contínuo de aeronaves no espaço aéreo de forma segura e manter os pilotos informados sobre eventos relevantes (informações do tempo, por exemplo). Para os pilotos, o sistema de CTA deve fornecer canais de comunicação confiáveis com os controladores e informações que auxiliem na condução da aeronave.

As aeronaves voam no espaço aéreo nas chamadas aerovias. O espaço aéreo é dividido em setores. Cada setor é composto de uma ou mais interseções. As interseções são conectadas através das linhas imaginárias chamadas aerovias. Em geral, existe pelo menos um centro de controle para cada setor, e cada centro de controle é formado por uma equipe de controladores. A Figura 31 ilustra como o espaço aéreo pode ser modelado.

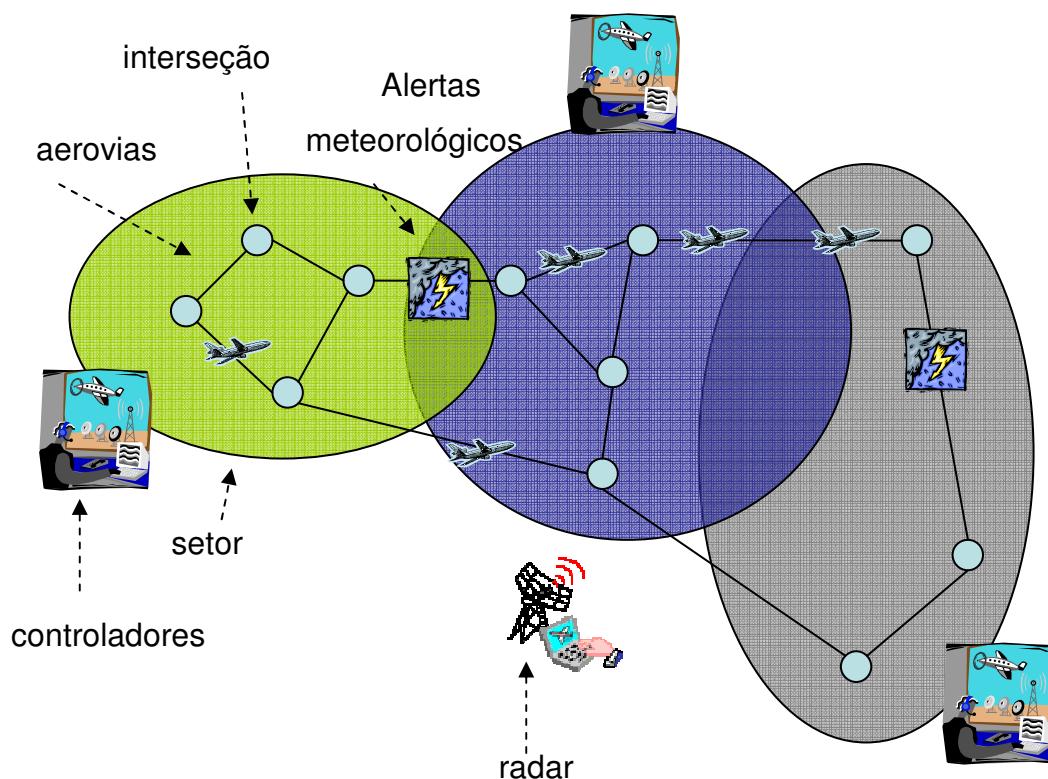


Figura 31 – Modelagem do Espaço Aéreo em um CTA

Tipicamente, existem quatro fases durante o movimento de uma aeronave: (i) controle de pista; (ii) decolagem; (iii) vôo e (iv) aterrissagem. Cada uma destas etapas é descrita em mais detalhes a seguir.

- i) Controle de pista – esta é a primeira etapa de vôo e, nela, o piloto se comunica com os controladores que informa como a aeronave deve se locomover em pista para efetuar a decolagem. Esta etapa é muito importante para evitar acidentes em solo, como o ocorrido em Tenerife – Espanha em 1977 (Wikipedia 2007). Nesta ocasião, uma grande neblina e a não obediência dos pilotos às ordens do controlador provocaram o maior acidente em número de mortos da história, através do choque entre uma aeronave tentando decolar, com outra aeronave ainda em solo, na mesma pista usada para a decolagem. Nesta etapa, as principais atividades são:
 - a. piloto apresenta plano de vôo;
 - b. controlador aprova ou rejeita o plano de vôo;
 - c. caso o plano de vôo tenha sido aprovado, o piloto informa a intenção de decolar;
 - d. controlador informa as etapas para o posicionamento da aeronave;
- ii) Decolagem – após a fase anterior, a aeronave estará posicionada para a decolagem. Nesta etapa o (a) piloto solicita autorização para decolar e (b) o controlador concede ou rejeita a autorização para decolagem.
- iii) Vôo – durante o vôo, as aeronaves são monitoradas por radares que disponibilizam informações como posicionamento, altitude e a velocidade. Além disso, a própria aeronave se comunica diretamente com os controladores enviando informações de sensores na aeronave. As aeronaves voam nas aerovias sob o monitoramento do controlador responsável pelo setor da aerovia. A medida que a aeronave segue o seu trajeto, ela pode cruzar vários setores. Entre setores vizinhos existe sempre uma área de interseção. Logo, para mudar de setor, a aeronave entra em um modo de operação chamado *hand-over*. Este modo de operação estabelece protocolos de comportamento diferenciados em relação ao vôo normal.

- iv) Aterrissagem – nesta etapa, os controladores precisam se certificar de que os poucos obedecerão aos requisitos de distância mínima e de intervalo mínimo entre poucos consecutivos. Além disso, os pilotos precisam cumprir determinados procedimentos tal como o circuito de tráfego padrão, que estabelece um caminho específico para a aproximação das aeronaves na pista, de maneira que toda aeronave deve respeitar este caminho.

Em cada uma destas etapas existe um conjunto de regulamentações que os pilotos e controladores devem seguir (Ndovie 1994; Rogério 2007). Neste estudo de caso, foi selecionado um subconjunto destas regulamentações classificadas como relevantes sob a ótica de governança e fidedignidade.

7.4.Casos de Uso

Os casos de uso abaixo descrevem as funcionalidades principais de um sistema de CTA (SCTA). Os casos de uso não descrevem os fluxos de eventos, pois o objetivo não é construir um SCTA totalmente funcional, mas identificar as principais interações entre os agentes que compõem o sistema e então identificar os aspectos de governança e de fidedignidade. Desta forma, os casos de uso identificados se assemelham a uma lista de requisitos, que, se refinada, pode gerar o fluxo de eventos e outras características normalmente encontradas em casos de uso.

CASO DE USO	
Identificação	CDU001_Manter_Separação_Lateral_Vertical_Longitudinal
Descrição Sucinta	
<p>O controlador de vôo deve possuir ferramentas visuais para a identificação da distância lateral, vertical e longitudinal entre as aeronaves.</p> <p>De posse da visualização o sistema deve fornecer ferramentas que permitam que o controlador interaja com os pilotos para informar as instruções de posicionamento da aeronave.</p> <p>A interação deve ser baseada em protocolos padrão bem definidos com o intuito de evitar ambigüidades.</p>	
Atores	
1. Controlador de vôo	

CASO DE USO	
Identificação	CDU002_Receber_Informações_Estações_Metereológicas
Descrição Sucinta	
O sistema deve ser capaz de receber dados das diversas estações meteorológicas disponíveis. Estes dados devem poder ser apresentados ao controlador.	
Atores	
1. Controlador de vôo 2. Estação Meteorológica (sistema externo)	

CASO DE USO	
Identificação	CDU003_Informar_Condições_Tempo_Ao_Piloto
Descrição Sucinta	
Os dados recebidos de estações meteorológicas devem ser apresentados aos pilotos na medida em que eles precisarem da informação. Um piloto precisa de uma determinada informação meteorológica se a informação se relaciona com o seu plano de vôo ou quando o piloto solicita explicitamente a informação.	
Atores	
1. Piloto 2. Estação Meteorológica	

CASO DE USO	
Identificação	CDU004_Receber_Informações_Posicionamento_Velocidade
Descrição Sucinta	
Os radares fornecem informações sobre posicionamento e velocidade das aeronaves dentro do seu perímetro de alcance.	
O sistema deve ser capaz de receber informações destes radares.	
Atores	
1. Controlador de Vôo 2. Radar (sistema externo)	

CASO DE USO	
Identificação	CDU005_Informar_Posicionamento_Velocidade
Descrição Sucinta	
Os controladores necessitam de informação precisa sobre posicionamento e velocidade das aeronaves. Esta informação é originada, principalmente, através dos radares. Desta forma, o sistema deve ser capaz de exibir esta informação tanto aos controladores quanto aos pilotos.	
Atores	
1. Controlador de Vôo 2. Piloto	

CASO DE USO	
Identificação	CDU006_Receber_Plano_Vôo
Descrição Sucinta	
Durante a etapa de <i>controle de pista</i> descrita na Seção 7.3, o piloto envia o plano de vôo para que o controlador aprove. A aprovação do plano de vôo precisa respeitar a regra da autonomia mínima. Autonomia é o tempo total que um aeronave é capaz de voar, em velocidade de cruzeiro, baseado na quantidade de combustível que ela possui. Para a realização de um vôo, a autonomia mínima será:	
Da decolagem ao destino mais o tempo entre o destino e a alternativa, mais 45 minutos de reserva. Ou seja:	
A -> B -> C + 45 min.	
(ORI) (DEST) (ALT) (reserva)	
Desta forma, o sistema deve ser capaz de receber os planos de vôo dos pilotos e disponibilizá-los para que os controladores os aprovem ou rejeitem.	
Atores	
1. Piloto 2. Controlador de vôo	

CASO DE USO	
Identificação	CDU007_Permitir_Aprovacao_Planos_Voo
Descrição Sucinta	
O sistema deve fornecer uma interface que permita que o controlador analise o plano de vôo e emita a aprovação ou rejeição do plano submetido.	
Atores	
1. Controlador de vôo	

CASO DE USO	
Identificação	CDU008_Garantir_Segurança_Aterrissagem
Descrição Sucinta	
Existem várias regras de segurança que precisam ser seguidas para melhorar o nível de segurança nas aterrissagens. O sistema deve monitorá-las de forma a verificar se elas estão sendo seguidas. Tanto os controladores quanto os pilotos devem ser informados em caso de não cumprimento das regras.	
Atores	
1. Piloto	
2. Controlador de vôo	

CASO DE USO	
Identificação	CDU009_Informar_Proativamente_Mudança_Controlador
Descrição Sucinta	
Ao mudar de setor, as aeronaves também passam a ser controladas por controladores diferentes. Ou seja, existe uma troca de controladores. O sistema deve proativamente informar aos pilotos e controladores envolvidos a respeito da nova configuração. Os pilotos são informados de quem é o novo controlador, o controlador anterior passa a não controlar mais a aeronave e o novo controlador recebe a informação da aeronave que estará sob o seu controle.	
Atores	
1. Piloto	
2. Controlador de vôo	

CASO DE USO	
Identificação	CDU010_Monitorar_Ações_Piloto
Descrição Sucinta	
<p>O sistema deve ser capaz de monitorar e armazenar todas as ações do piloto em relação a pilotagem.</p> <p>Esta informação pode ser utilizada para auditoria ou mesmo para identificação de falhas em tempo de execução.</p>	
Atores	
1. Piloto	

CASO DE USO	
Identificação	CDU011_Informar_Pilotos_Ações
Descrição Sucinta	
<p>De posse das informações armazenadas com o monitoramento, o sistema eventualmente pode sugerir ao piloto quais ações são mais apropriadas dado que ele executou alguma ação fora do procedimento.</p> <p>Desta forma, o sistema além de exibir alertas, exibe também sugestões de procedimentos a serem adotados.</p>	
Atores	
1. Piloto	

CASO DE USO	
Identificação	CDU012_Prover_Comunicação_Não_Ambígua
Descrição Sucinta	
<p>Um dos grandes riscos de problemas em SCTA é o problema da comunicação. Dificuldade de entendimento entre pilotos e controladores que não possuem o idioma inglês como primeira língua resulta na adoção de procedimentos errados. Além disso, ambigüidades também são geradas devido a não utilização de comunicações padrão.</p> <p>Desta forma, o sistema deve prover o maior número possível de protocolos de comunicação bem definidos que estabeleçam as formas de interação entre controladores e pilotos.</p>	

Atores	
1. Controlador de vôo	
2. Piloto	

CASO DE USO	
Identificação	CDU013_Garantir_Segurança_Decolagem
Descrição Sucinta	
<p>É na etapa de decolagem que ocorrem aproximadamente 15,7% dos acidentes com causas não mecânicas (Krey 2006). Existe um conjunto de regras de segurança que devem ser seguidas por pilotos e controladores.</p> <p>O sistema deve verificar se estas regras estão sendo cumpridas e intervir em caso de não cumprimento.</p>	
Atores	
1. Piloto	
2. Controlador de vôo	

CASO DE USO	
Identificação	CDU014_Garantir_Segurança_Vôo
Descrição Sucinta	
<p>Aproximadamente 9,7% dos acidentes aéreos com causas não mecânicas ocorrem durante o vôo (Krey 2006). Porém, os acidentes durante o vôo são responsáveis por 22,8% do total de acidentes fatais.</p> <p>Esta estatística indica que garantir a segurança durante esta fase é crítico para diminuir o número de acidentes fatais. Portanto, o sistema deve continuamente monitorar as ações dos pilotos, as informações da aeronave e as instruções dos controladores e intervir quando alguma situação não ocorrer conforme o especificado.</p>	
Atores	
1. Controlador de vôo	
2. Piloto	

CASO DE USO	
Identificação	CDU015_Exibir_Interface_Gráfica
Descrição Sucinta	
O sistema deve se comunicar com o controlador e com o piloto através de interfaces gráficas.	
Atores	
1. Controlador de vôo	
2. Piloto	

7.5.Arquitetura

Um SCTA é composto de vários subsistemas que precisam cooperar para atingir o objetivo do sistema. Cada subsistema possui um objetivo específico para atingir. Por exemplo, o subsistema TACS possui como objetivo evitar colisões entre as aeronaves em vôo. Além disso, os subsistemas também possuem autonomia de decisão, muito embora esta autonomia possa ser na maioria dos casos supervisionada por atores humanos que podem interferir no processo. Além disso, muitos destes subsistemas estão geograficamente distribuídos e precisam se comunicar via protocolos de rede. Neste trabalho, optou-se por representar cada um destes subsistemas por um agente de software conforme pode ser visto na arquitetura apresentada Figura 32. Os elementos desta arquitetura são descritos a seguir:

- Estação Meteorológica – fornece informações sobre as condições do tempo atuais e futuras.
- Radar – monitora o posicionamento e velocidade das aeronaves no espaço aéreo.
- Piloto – um agente de software que representa o piloto humano.
- Controlador – agente de software que representa o controlador humano.
- ATC GUI – Interface gráfica que permite com que o controlador tenha acesso a todas as informações para basear a tomada de decisões.
- ATC Façade – Agente que atua como um “broker” da comunicação entre os sistemas descritos acima e os agentes que fazem parte da organização de agentes composta pelos agentes descritos a seguir. De

fato, este agente é uma simplificação do sistema para efeitos de prototipação.

- TCAS – agente que implementa um sistema de alertas de possíveis rotas de colisão entre aeronaves.
- CKPT – agente responsável pelas informações do cockpit da aeronave.
- ADMG – gerencia as etapas de decolagem e pouso.
- MSAW – verifica constantemente violações de segurança relacionadas à altitude permitida.
- SYSC – auxilia controladores e pilotos na etapa de “hand-over”, ou seja, no momento em que uma aeronave muda de setor.

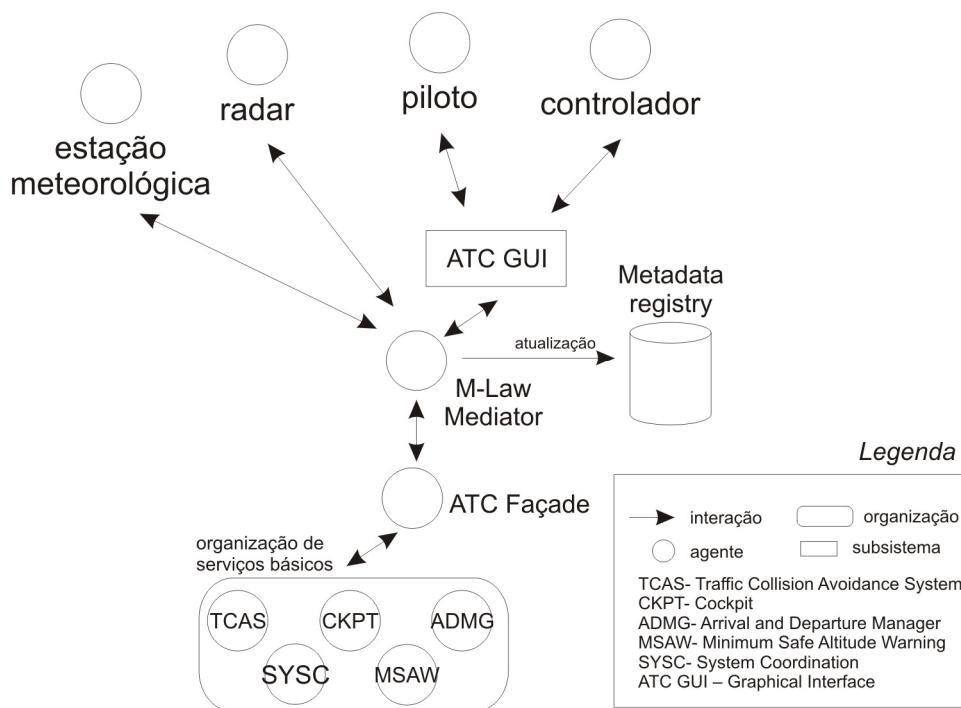


Figura 32 – Arquitetura do Estudo de Caso

Nas seções 7.6 e 7.7, o estudo de caso é analisado sob as óticas de governança e de fidedignidade. Utiliza-se a abordagem desta tese para representar as leis sob estes dois pontos de vista.

7.6.Análise da Governança

Para a realização da análise de forma sistemática, o problema será modelado através dos passos descritos na Tabela 12.

- | |
|--|
| Passo 1: identificar as cenas de interação |
| Passo 2: identificar os protocolos de interação de cada cena |
| Passo 3: identificação dos outros elementos de leis |

Tabela 12 – Guia de Análise de Governança

7.6.1.Passo 1

Conforme identificado na Seção 7.3, existem 4 fases principais em um vôo: controle de pista, decolagem, vôo e aterrissagem. Cada uma destas fases possui protocolos de comunicação e regras bem delimitadas. Estas fases são modeladas como cenas.

7.6.2.Passo 2 e 3

A identificação dos protocolos é realizada a partir da análise das interações entre os agentes em cada cena.

Cena 1: controle de pista. Nesta etapa, o piloto apresenta o plano de vôo e o controlador aprova ou rejeita o plano de vôo. Caso o plano de vôo tenha sido aprovado, o piloto informa a intenção de decolar e por fim, o controlador informa as etapas para o posicionamento da aeronave. A Figura 33 mostra o protocolo de interação destas atividades. A representação deste protocolo em XMLaw é ilustrada na listagem de Código 38.

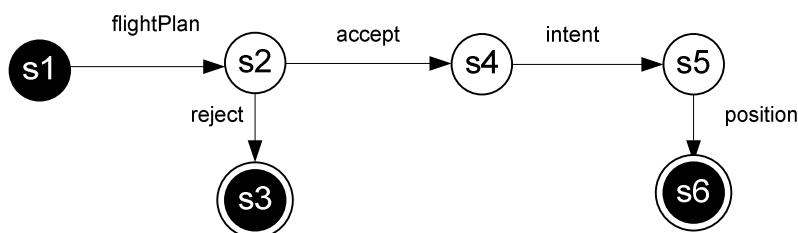


Figura 33 – Protocolo de Interação da Cena Controle de Pista

```

groundControl{ //nome da cena

// Mensagens
flightPlan{pilot, controller,propose(flightPlan($content))}
reject{controller, pilot, reject-proposal}
accept{controller, pilot, accept-proposal}
intent{pilot, controller, request(go-position)}
position{controller, pilot, inform(pos, $instructions)}

// Estados especiais
s1{initial}
s3{failure}
s6{success}

// Transições
t1{s1->s2, flightPlan}
t2{s2->s3, reject}
t3{s2->s4, accept}
t4{s4->s5, intent}
t5{s5->s6, position}
}

```

Código 38 – Protocolo de interação em XMLaw da cena *groundControl*

Cena 2: decolagem – o protocolo de interação desta etapa é bastante simples. O piloto solicita autorização para decolar e o controlador concede ou rejeita a autorização para decolagem.

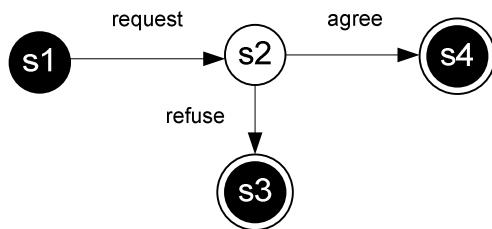


Figura 34 - Protocolo de Interação da Cena Decolagem

```

take-off{ //nome da cena

// Mensagens
request{pilot, controller, request(take-off)}
refuse{controller, pilot, refuse }
agree{controller, pilot, agree }

// Estados especiais
s1{initial}
s3{failure}
s4{success}

// Transições
t1{s1->s2, request}
t2{s2->s3, refuse}
t3{s2->s4, agree}
}

```

Código 39 – Protocolo de interação em XMLaw da cena *take-off*

Cena 3: vôo – nesta etapa, existem 2 tipos de interações principais: monitoramento da aeronave e o processo de mudança de setor (*hand-over*). A Figura 35 mostra o protocolo de interação. A partir do estado inicial, duas mensagens podem ser enviadas: *progressR* e *progressA*. A mensagem *progressR* representa uma mensagem enviada do radar para o controlador informando o posicionamento e a velocidade da aeronave. Algumas aeronaves são equipadas com dispositivos que também permitem que a própria aeronave envie uma mensagem para o controlador informando o posicionamento e a velocidade da aeronave. Esta mensagem é representada pela mensagem *progressA*. No estado *s2*, existem 3 mensagens possíveis: *switch*, *progressR* e *progressA*. Elas representam o vôo da aeronave, onde constantemente existem mensagens de monitoramento e no momento que a aeronave for realizar o *hand-over* ocorrerá a mensagem *siwtch*. Finalmente, a cena de vôo se encerra quando o piloto enviar uma mensagem de intenção de pouso (*landing*) para o controlador. O Código 40 mostra a representação deste protocolo em XMLaw.

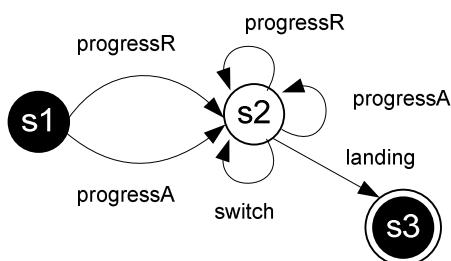


Figura 35 - Protocolo de Interação da Cena Vôo

```

flight{ // nome da cena

// Mensagens
progressR{radar, controller, inform(strip,
$flightProgressStrip)}
progressA{pilot, controller, inform(strip,
$flightProgressStrip)}
switch{controller, pilot, inform(switch, $newController)}
landing{pilot, controller, inform(landIntention)}

// Estados especiais
s1{initial}
s3{success}

// Transições
t1{s1->s2, progressR}
t2{s1->s2, progressA}
t3{s2->s2, progressR}
t4{s2->s2, progressA}
  
```

```
t5{s2->s2, switch}
t6{s2->s3, landing}
}
```

Código 40 – Protocolo de Interação em XMLaw da cena *flight*

Cena 4: aterrissagem – o protocolo de interação desta etapa consiste no piloto solicitar permissão para pouso e aguardar a permissão do controlador. A permissão pode ser concedida imediatamente ou após algum tempo de espera.

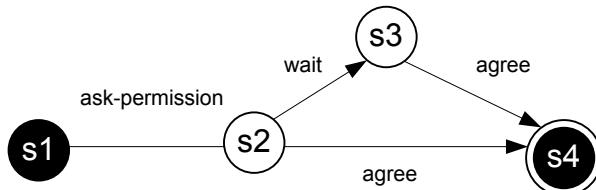


Figura 36 - Protocolo de Interação da Cena Aterrissagem

```

landing{
  ask-permission{pilot, controller,
  request(landingPermission)}
  wait{controller, pilot, inform(wait)}
  agree{controller, pilot, agree(permission)}

  s1{initial}
  s4{success}

  t1{s1->s2, ask-permission}
  t2{s2->s3, wait}
  t3{s3->s4, agree}
  t4{s2->s4, agree}

}
  
```

Durante a identificação dos protocolos acima, foi possível perceber quais eram os agentes que interagem em cada cena. São eles:

- Cena *groundControl*: Pilot e Controller
- Cena *take-off*: Pilot e Controller
- Cena *flight*: Pilot, Radar e Controller
- Cena *landing*: Pilot e Controller

7.7.Análise da Fidedignidade

A análise da fidedignidade envolve a análise das ameaças, a identificação de estratégias de mitigação destas ameaças, a introdução de um agente detector de falhas e a especificação de leis que implementam as estratégias de mitigação.

7.7.1. Análise das ameaças

Nesta etapa, procura-se identificar, priorizar e relacionar as ameaças que, se efetivadas, poderiam prejudicar o objetivo do sistema. A cada ameaça atribui-se um identificador único, uma lista de identificadores de casos de uso, uma descrição e a consequência da sua efetivação. O identificador é utilizado para fazer um rastreamento da ameaça no restante da documentação. Os identificadores de casos de uso são utilizados para identificar os casos de uso que a ameaça pode afetar. A descrição fornece um breve contexto para a ameaça e, por fim, a consequência identifica o que acontece em uma situação que a ameaça se concretiza. O objetivo deste campo é identificar a criticalidade que é definida em uma escala de 1 a 5.

Embora, a Tabela 13 utilizada para a elicitação das ameaças utilizadas neste trabalho seja bastante simples, ela atende as necessidades deste estudo de caso. Caso o problema a ser resolvido demande por uma análise de ameaças mais refinada, é possível utilizar-se uma das várias abordagens existentes na literatura para elicitação, análise e gerenciamento de riscos (Boehm 1981; Charette 1989; Karolak 1996).

Id	Ids CDUs	Descrição	Consequências
CDL01	CDU01	Aeronave se aproxima demais de outra aeronave	4 Colisão
CDL02	CDU14	Aeronave voa em uma altitude diferente da planejada	4 Colisão
CDL03	CDU13	Piloto não ativa o "flap" no momento da decolagem	4 Queda
CDL04	CDU12	Ocorre falha no entendimento da comunicação entre o piloto e a torre de controle	3 Colisão, informação inconsistente
CDL05	CDU13	O piloto não ativa o sistema descongelador	3 Congelamento da turbina

CDL06	CDU14	Piloto desativa inapropriadamente um instrumento	2 Informação inconsistente, queda, dificuldade de auditoria
CDL07	CDU02	Falha na comunicação com a estação meteorológica	2 Turbulência, queda
CDL08	CDU04	Falha na comunicação com os radares	5 Colisão
CDL09	CDU07	Controlador aprova plano de vôo fora das especificações de segurança	3 Acidente
CDL10	CDU07	Controlador concede permissão de vôo que acarretará em uma distância de autonomia mínima menor que o especificado. A autonomia mínima é definida como sendo o tempo total que um aeronave é capaz de voar, em velocidade de cruzeiro, baseado na quantidade de combustível que ela possui. Para a realização de um vôo, a autonomia mínima será: Da decolagem ao destino mais o tempo entre o destino e a alternativa, mais 45 minutos de reserva. Ou seja: A -> B -> C + 45 min. (DEP) (ARR) (ALT) (reserva)	2 Falta de combustível
CDL11	CDU09	Controlador perde o canal de comunicação	5 Acidentes
CDL12	CDU08	Para que uma aeronave possa operar sem restrições em uma determinada	1 Prejuízos na pista

		pista, o ACN da aeronave deverá ser menor ou igual que o PCN da pista (ACN <= PCN). A ameaça consiste em a aeronave solicitar pouso com ACN>PCN	
CDL13	CDU08	O piloto não executa a operação padrão de circuito de tráfego ao pousar. A altura padrão para as aeronaves realizarem o circuito de tráfego é: <ul style="list-style-type: none"> • 1500ft (pés) para aeronaves a jato; • 1000ft (pés) para aeronaves a hélice. Todas as curvas são feitas para a esquerda	2 Colisão
CDL14	CDU14	Exceto em procedimentos de pouso e decolagem, as aeronaves não poderão voar sobre cidades, povoados, lugares habitados ou grupo de pessoas ao ar livre a uma altura inferior a 1000 pés (300M) acima do obstáculo mais alto existente num raio de 600M em torno da acft; A ameaça consiste no piloto descumprir estas restrições.	3 Colisão em prédios, paraquedistas, etc.
CDL15	CDU14	Exceto em procedimentos de pouso e decolagem, as aeronaves não poderão voar em lugares desabitados em altura inferior a 500 pés (150M) sobre o solo ou água. A ameaça consiste no piloto voar a uma altura inferior a 500 pés.	3 Colisão com formações geológicas

Tabela 13 – Elicitação das Ameaças

Ao se concretizarem, as ameaças afetam os atributos de fidedignidade do sistema. Na Seção 7.7.2, mostra-se como o XMLaw e o M-Law são utilizados para especificar e implementar estratégias de mitigação destas ameaças.

7.7.2. Mitigação das Ameaças

Ameaça	Mitigação
CDL01	<ul style="list-style-type: none"> - solicitar plano de ação ao agente TCAS (<i>traffic collision avoidance system</i>); - informar plano de ação ao piloto; - informar plano de ação ao controlador.
CDL02	<ul style="list-style-type: none"> - solicitar plano de ação ao agente MSAW (<i>minimum safe altitude warning</i>); - informar plano de ação ao piloto; - informar plano de ação ao controlador.
CDL03	<ul style="list-style-type: none"> - emitir alerta ao piloto.
CDL04	<ul style="list-style-type: none"> - utilizar protocolos de comunicação bem definidos e mensagens padronizadas.
CDL05	<ul style="list-style-type: none"> - emitir alerta ao piloto.
CDL06	<ul style="list-style-type: none"> - Alertar o piloto das consequências de se permanecer com o instrumento desligado.
CDL07	<ul style="list-style-type: none"> - Solicitar ao agente ATC <i>Facade</i> outra estação meteorológica que possa prover os dados.
CDL08	<ul style="list-style-type: none"> - Alertar administradores do sistema através de email.
CDL09	<ul style="list-style-type: none"> - Solicitar ao ADMG (<i>arrival and departure manager</i>) que alerte o controlador, informe onde está o erro e explique as consequências. - Solicitar confirmação de ação.
CDL10	<ul style="list-style-type: none"> - Solicitar ao ADMG (<i>arrival and departure manager</i>) que alerte o controlador, informe onde está o erro e explique as consequências. - Solicitar confirmação de ação
CDL11	<ul style="list-style-type: none"> - Solicitar ao ATC <i>Facade</i> que avise aos pilotos quais são os controladores disponíveis.
CDL12	<ul style="list-style-type: none"> - Solicitar ao ADMG (<i>arrival and departure manager</i>) que alerte o controlador, informe onde está o erro e explique as consequências.

	<ul style="list-style-type: none"> - Solicitar confirmação de ação.
CDL13	<ul style="list-style-type: none"> - Alertar o controlador e o piloto sobre o descumprimento do circuito de tráfego padrão.
CDL14	<ul style="list-style-type: none"> - solicitar plano de ação ao agente MSAW (<i>minimum safe altitude warning</i>); - informar plano de ação ao piloto; - informar plano de ação ao controlador
CDL15	<ul style="list-style-type: none"> - solicitar plano de ação ao agente MSAW (<i>minimum safe altitude warning</i>); - informar plano de ação ao piloto; - informar plano de ação ao controlador

Tabela 14 – Estratégias de Mitigação das Ameaças

Uma vez identificada as estratégias de mitigação é preciso alterar a especificação das leis definidas na Seção 7.6 para contemplar a implementação destas estratégias. Para que as estratégias de mitigação possam ser implementadas é necessário que se possa identificar a ocorrência da ameaça durante a execução do sistema. A abordagem de governança proposta nesta tese faz isso através da especificação da lei para detectar a ameaça e através de um agente especial intitulado “detector de falhas”. Este agente identifica os agentes que estão indisponíveis no sistema. Esta indisponibilidade pode ter sido provocada pelo excesso de processamento, pelo não funcionamento do agente ou até mesmo por problemas no link de comunicação. O objetivo deste agente é reduzir o número de clocks com a finalidade de identificar agentes indisponíveis. Ao detectar um agente indisponível, o *detector de falhas* emitirá um evento no XMLaw *agent_unavailable* que pode ser utilizado para ativar algum outro elemento do XMLaw.

7.7.3. Agente Detector de Falhas

Neste trabalho, optou-se por implementar o agente detector de falhas através da utilização de um agente único que implementa um algoritmo de *heartbeat*. Este algoritmo consiste em enviar mensagens de controle periodicamente para os agentes do sistema esperando que os agentes respondam a esta mensagem. Devido

ao grande número de variáveis e a simplicidade da estratégia de implementação do *heartbeat*, podem ocorrer falsos negativos, ou seja, a demora da resposta pode ser interpretada como indisponibilidade do agente. Sendo assim, para minimizar este problema, o agente tentará pelo menos 3 vezes antes de indicar que o agente está indisponível. Se ainda assim, esta estratégia não for suficiente para um determinado domínio de aplicação, então o agente detector poderia ser substituído por implementações mais sofisticadas, como por exemplo o *Globus Heartbeat Monitor* (Stelling, DeMatteis et al. 1999). Na Figura 37, mostra-se a arquitetura do estudo de caso modificada para a inclusão do agente detector de falhas.

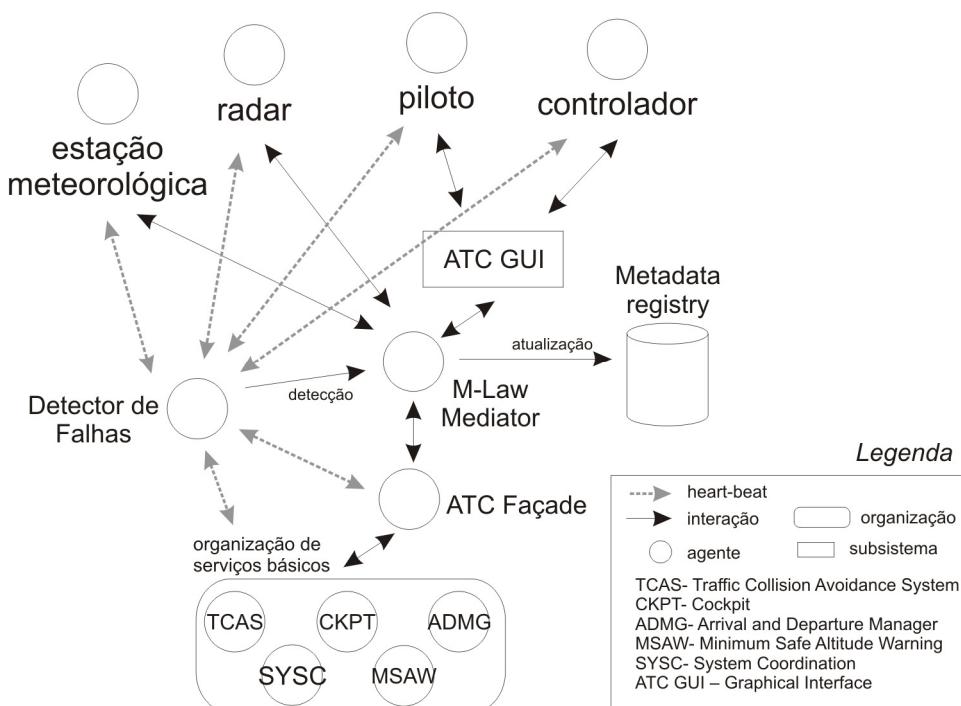


Figura 37 – Arquitetura do Estudo de Caso com a Inclusão do Agente Detector de Falhas

7.7.4.Modificação das Leis para Implementar as Ameaças

Nesta seção, as estratégias de mitigação de cada ameaça serão especificadas utilizando o XMLaw. Para isto, apresenta-se o identificador da ameaça, o contexto de lei no qual a ameaça afeta, a estratégia de detecção da ameaça e finalmente, quando for necessário, o exemplo de código XMLaw da implementação.

Ameaça: CDL01
Contexto: Cena de Vôo
Estratégia de detecção: para detectar se as aeronaves estão próximas, é preciso

saber os seus posicionamentos. O posicionamento é informado pelas mensagens *progressR* ou *progressA*. Baseado nestas mensagens, a lei deve executar um algoritmo de cálculo de distância. Se este algoritmo indicar uma distância menor do que o permitido, então o sistema de prevenção de colisão deve ser alertado. O algoritmo de cálculo e verificação de distância é executado por uma *constraint*. Esta *constraint* retorna *true* quando a distância é menor que o determinado. Quando uma *constraint* retorna *true*, o evento *constraint_not_satisfied* é gerado. Este evento deve ativar uma *action* que avisa o TCAS. Além disso, de acordo com a estratégia de mitigação especificada na Tabela 14, o piloto e o controlador também devem ser informados. O Código 41 apresentado a seguir mostra a cena de vôo modificada para a inclusão da implementação desta estratégia de mitigação, e no Código 42 apresenta-se a implementação da *constraint checkDistance*.

```

flight{ // nome da cena

    // Mensagens
    progressR{radar, controller, inform(strip,
$flightProgressStrip)}
    progressA{pilot, controller, inform(strip,
$flightProgressStrip)}
    switch{controller, pilot, inform(switch, $newController)}
    landing{pilot, controller, inform(landIntention)}

    // Estados especiais
    s1{initial}
    s3{success}

    // Transições
    t1{s1->s2, progressR, [checkDistance]}
    t2{s1->s2, progressA, [checkDistance]}
    t3{s2->s2, progressR, [checkDistance]}
    t4{s2->s2, progressA, [checkDistance]}
    t5{s2->s2, switch}
    t6{s2->s3, landing}

    // Constraints
    checkDistance{br.les.CheckDistance}

    // Actions
    warnTcas{ ((checkDistance, constraint_not_satisfied)),
br.les.WarnTCAS}
    warnPilot{ ((checkDistance, constraint_not_satisfied)),
br.les.WarnPilot}
    warnController{
((checkDistance, constraint_not_satisfied)),
br.les.WarnController}
}

```

Mitigação
da ameaça
CDL01

Código 41 – XMLaw da cena de vôo com a estratégia de mitigação da

Ameaça CDL01

```

class CheckDistance implements IConstraint{
    private void init(){
        ...
    }
    public boolean constrain(ReadonlyContext ctx){
        String id = ctx.get("flightProgressStrip.airplaneId");
        String x = ctx.get("flightProgressStrip.posX");
        String y = ctx.get("flightProgressStrip.posY");
        String z = ctx.get("flightProgressStrip.posZ");

        saveCurrentPosition(id,x,y,z);

        if ( tooClose(id) ) {
            return true;
        }
    }
}

```

Código 42 – Implementação da Constraint CheckDistance

Ameaça: CDL02

Contexto: Cena de Vôo

Estratégia de detecção: utilizar as mensagens *progressR* ou *progressA* como informações para uma constraint que verifica a altura segura. Esta *constraint* retorna *true* quando a altura é menor que o determinado. A constraint ativa uma *action* que avisa o MSAW. Além disso, o piloto e o controlador também devem ser informados. O Código 43 apresentado a seguir mostra a cena de vôo modificada para a inclusão da implementação desta estratégia de mitigação. As setas apontando para retângulos tracejados indicam onde foram feitas as modificações. A implementação da *constraint* foi omitida, pois é bastante similar a *constraint* exibida no Código 42.

```

flight{ // nome da cena

    // Mensagens
    progressR{radar, controller, inform(strip,
$flightProgressStrip)}
    progressA{pilot, controller, inform(strip,
$flightProgressStrip)}
    switch{controller, pilot, inform(switch, $newController)}
    landing{pilot, controller, inform(landIntention)}

    // Estados especiais
}

```

```

s1{initial}
s3{success}

// Transições
t1{s1->s2, progressR, [checkDistance, checkAltitude] }
t2{s1->s2, progressA, [checkDistance, checkAltitude] }
t3{s2->s2, progressR, [checkDistance, checkAltitude] }
t4{s2->s2, progressA, [checkDistance, checkAltitude] }
t5{s2->s2, switch}
t6{s2->s3, landing}

// Constraints
checkDistance{br.les.CheckDistance},
checkAltitude{br.les.CheckAltitude} ←

// Actions
warnTcas{ ((checkDistance, constraint_not_satisfied)),
br.les.WarnTCAS}

warnMsaw{ ( (checkAltitude, constraint_not_satisfied) ),
br.les.WarnMSAW} ←

warnPilot{
(
    (checkDistance, constraint_not_satisfied),
    (checkAltitude, constraint_not_satisfied) ) ←
), br.les.WarnPilot}

warnController{
(
    (checkDistance, constraint_not_satisfied),
    (checkAltitude, constraint_not_satisfied) ) ←
), br.les.WarnController}
}

```

Código 43 – XMLaw da Cena de Vôo com a Estratégia de Mitigação da Ameaça CDL02

Ameaça: CDL03

Contexto: Cena de decolagem

Estratégia de detecção: a informação de que o piloto ativou ou não o *flap* está disponível somente na própria aeronave. Desta forma, para monitorar esta informação, acrescenta-se uma mensagem de status da aeronave enquanto o piloto estiver decolando. Esta mensagem deve ser enviada continuamente até que o piloto inicie a etapa de vôo. Desta forma, uma *action* pode ser ativada a cada novo reporte de status. Esta *action* verifica se o *flap* foi ativado e caso não tenha sido, emite um alerta para o piloto.

```

take-off{ //nome da cena

// Mensagens
request{pilot, controller, request(take-off)}
refuse{controller, pilot, refuse }
agree{controller, pilot, agree }
status{pilot, controller, inform($status)}
flight{pilot, controller, inform(flightPhase)} } ←

// Estados especiais
s1{initial}
s3{failure}
s5{success} } ←

// Transições
t1{s1->s2, request}
t2{s2->s3, refuse}
t3{s2->s4, agree}
t4{s4->s4, status}
t5{s4->s5, flight} } ←

// Actions
warnPilot{ (t4), br.les.WarnPilot} } ←
}

```

Código 44 – XMLaw da Cena de Decolagem com a Estratégia de Mitigação da Ameaça CDL03

Ameaça: CDL04

Contexto: Todas as cenas

Estratégia de detecção: as falhas de comunicação podem ser consideravelmente reduzidas quando se utilizam protocolos de interação bem definidos. Neste caso, a estratégia é utilizar os protocolos de interação do próprio XMLaw para definir as comunicações que são válidas. Os protocolos das cenas já foram apresentados no decorrer deste capítulo.

Ameaça: CDL05

Contexto: Cena de vôo

Estratégia de detecção: para a detecção de que o piloto não ativou o sistema descongelador utilizou-se uma abordagem similar a estratégia utilizada para a detecção da ameaça CDL03. A *action* *warnDefroster* é ativada a cada novo relatório de status. Esta *action* verifica se o descongelador foi ativado e caso não tenha sido, emite um alerta para o piloto. Nota-se que esta estratégia foi diferente da utilizada para identificar a distância e altitude. Nestas duas, utilizou-se uma *constraint*. Entretanto, nada impede que se tivesse utilizado a estratégia de *action* proposta nessa ameaça.

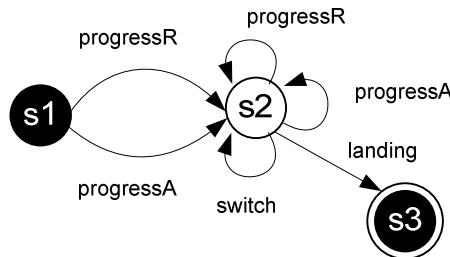


Figura 38 – Protocolo de Interação da cena de vôo

```

flight{ // nome da cena

  // Mensagens
  progressR{radar, controller, inform(strip,
$flightProgressStrip)}
  progressA{pilot, controller, inform(strip,
$flightProgressStrip)}
  switch{controller, pilot, inform(switch, $newController)}
  landing{pilot, controller, inform(landIntention)}

  // Estados especiais
  s1{initial}
  s3{success}

  // Transições
  t1{s1->s2, progressR, [checkDistance, checkAltitude]}
  t2{s1->s2, progressA, [checkDistance, checkAltitude]}
  t3{s2->s2, progressR, [checkDistance, checkAltitude]}
  t4{s2->s2, progressA, [checkDistance, checkAltitude]}
  t5{s2->s2, switch}
  t6{s2->s3, landing}

  // Constraints
  checkDistance{br.les.CheckDistance}
  checkAltitude{br.les.CheckAltitude}
}
  
```

```
// Actions
warnTcas{ ((checkDistance,constraint_not_satisfied)),
br.les.WarnTCAS}

warnMsaw{ ( (checkAltitude,constraint_not_satisfied) ),
br.les.WarnMSAW}

warnPilot{
(
    (checkDistance,constraint_not_satisfied),
    (checkAltitude,constraint_not_satisfied)
), br.les.WarnPilot}

warnController{
(
    (checkDistance,constraint_not_satisfied),
    (checkAltitude,constraint_not_satisfied)
),
br.les.WarnController}

-----[ warnDefroster{ (t1,t2,t3,t4), br.les.WarnDefroster} ]----- ←
```

Código 45 – XMLaw da Cena de Vôo com a Estratégia de Mitigação da Ameaça CDL05

Ameaça: CDL06

Contexto: Cena de vôo

Estratégia de detecção: A *action checkInstruments* é ativada a cada nova comunicação de status. Esta *action* verifica se o algum instrumento foi desativado durante o vôo e caso tenha sido, informa ao piloto as consequências de ter o instrumento desligado.

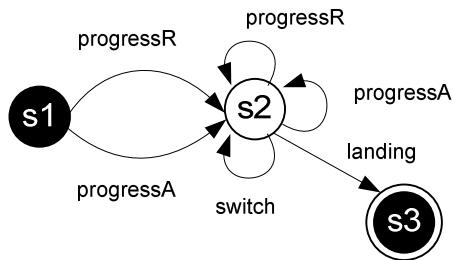


Figura 39 – Protocolo de Interação da Cena de Vôo

```

flight{ // nome da cena

// Mensagens
  progressR{radar, controller, inform(strip,
$flightProgressStrip)}
  progressA{pilot, controller, inform(strip,
$flightProgressStrip)}
  switch{controller, pilot, inform(switch, $newController)}
  landing{pilot, controller, inform(landIntention)}

// Estados especiais
  s1{initial}
  s3{success}

// Transições
  t1{s1->s2, progressR, [checkDistance, checkAltitude]}
  t2{s1->s2, progressA, [checkDistance, checkAltitude]}
  t3{s2->s2, progressR, [checkDistance, checkAltitude]}
  t4{s2->s2, progressA, [checkDistance, checkAltitude]}
  t5{s2->s2, switch}
  t6{s2->s3, landing}

// Constraints
  checkDistance{br.les.CheckDistance}
  checkAltitude{br.les.CheckAltitude}

// Actions
  warnTcas{ ((checkDistance, constraint_not_satisfied)), 
br.les.WarnTCAS}

  warnMsaw{ ( (checkAltitude, constraint_not_satisfied) ), 
br.les.WarnMSAW}
  
```

```

warnPilot{
    (
        (checkDistance,constraint_not_satisfied),
        (checkAltitude,constraint_not_satisfied)
    ), br.les.WarnPilot}

warnController{
    (
        (checkDistance,constraint_not_satisfied),
        (checkAltitude,constraint_not_satisfied)
    ), br.les.WarnController}

warnDefroster{ (t1,t2,t3,t4), br.les.WarnDefroster}

[ checkInstruments{ ( t1,t2,t3,t4), br.les.CheckInstruments} ]

}

```

Código 46 – XMLaw da Cena de Vôo com a Estratégia de Mitigação da Ameaça CDL06

Ameaça: CDL07

Contexto: Todas as cenas

Estratégia de detecção: a falha de comunicação com a estação meteorológica poderia ser detectada de duas formas: através do uso combinado do protocolo de interação e de um *clock* ou através do agente detector de falhas. Optou-se pelo uso do agente que informará caso ele se depare com 3 tentativas mal-sucedidas de comunicação com a estação meteorológica. Neste caso, o agente gera o evento *agent_unavailable* que é captado por uma *action*, que informa o agente ATC para utilizar outra estação meteorológica. Neste caso, a *action* é declarada em um escopo global. Desta forma, ela é válida para todas as cenas.

```

atc-law{
    // global_actions
    [ switchStation{(failure_detector, agent_unavailable),
    br.les.SwitchStation} ] }

    //cena
    groundControl {
        ...
    }

    //cena
    take-off{
        ...
    }

    //cena
    flight{
}

```

```

    ...
}

//cena
landing{
    ...
}

}//end law

```

Código 47 – XMLaw Utilizando o Agente Detector de Falhas

Ameaça: CDL08

Contexto: Todas as cenas

Estratégia de detecção: no caso de falha de comunicação com os radares, a estratégia de mitigação adotada consiste em enviar um email para os administradores do sistema. A falha é detectada através do agente detector de falhas, que através do evento *agent_unavailable* ativa uma *action* de envio de email.

```

atc-law{
    // global actions
    switchStation{(failure_detector, agent_unavailable),
    br.les.SwitchStation}

    radarDown{(failure_detector, agent_unavailable),
    br.les.RadarDown} ----->

    //cena
    groundControl {
        ...
    }

    //cena
    take-off{
        ...
    }

    //cena
    flight{
        ...
    }

    //cena
    landing{
        ...
    }

}//end law

```

Código 48 – Utilização do Agente Detector de Falhas para Reportar Falhas de Comunicação com os Radares.

Ameaça: CDL09

Contexto: Cena de controle de pista

Estratégia de detecção: foram utilizados vários elementos do XMLaw de forma combinada (Código 49). Declara-se uma *constraint* na transição *t3*. Essa *constraint* verifica se o plano de vôo está de acordo com as regras. Caso não esteja, a *constraint* gera o evento *constraint_not_satisfied* e evita que a transição *t3* dispare. Porém, o evento de *constraint_not_satisfied* faz com que a transição *t6* dispare. A transição *t6* muda o estado do protocolo de *s2* para *s7*. Por sua vez, disparo da transição *t6* faz com que a ação *askConfirmation* seja ativada. Esta ação envia uma mensagem para o controlador solicitando a confirmação da aprovação do plano de vôo, visto que algo fora dos padrões fora encontrado. Neste ponto o protocolo está no estado *s7* e só existem duas transições de saída. A transição *t8* dispara quando o controlador rejeita o plano e a transição *t7* dispara quando o controlador confirma a aprovação do plano de vôo.

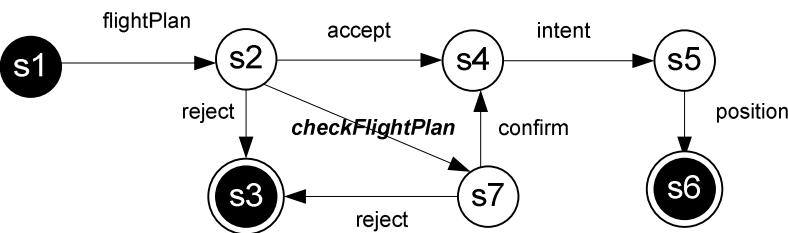


Figura 40 – Protocolo de Interação da Cena de Controle de Pista

```

groundControl{ // nome da cena

// Mensagens
flightPlan{pilot,
controller,propose(flightPlan($content))}
reject{controller, pilot, reject-proposal}
accept{controller, pilot, accept-proposal}
intent{pilot, controller, request(go-position)}
position{controller, pilot, inform(pos, $instructions)}
confirm{controller, pilot, confirm(accept-proposal)}

// Estados especiais
s1{initial}
s3{failure}
s6{success}

// Transições
t1{s1->s2, flightPlan}
t2{s2->s3, reject}
t3{s2->s4, accept, [checkFlightPlan]}
t4{s4->s5, intent}
t5{s5->s6, position}
t6{s2->s7, (checkFlightPlan, constraint_not_satisfied)}
t7{s7->s4, confirm}
t8{s7->s3, reject}

// constraints
checkFlightPlan{br.les.CheckFlightPlan}

// action
askConfirmation{t6, br.les. AskConfirmation}

}
  
```

O código XMLLaw define o protocolo de interação. Ele inclui mensagens para 'flightPlan', 'reject', 'accept', 'intent', 'position' e 'confirm'. Os estados especiais são 's1' (initial), 's3' (failure) e 's6' (success). As transições são: t1 (s1 para s2 com 'flightPlan'); t2 (s2 para s3 com 'reject'); t3 (s2 para s4 com 'accept' e constraint 'checkFlightPlan'); t4 (s4 para s5 com 'intent'); t5 (s5 para s6 com 'position'); t6 (s2 para s7 com constraint 'checkFlightPlan' e constraint 'constraint_not_satisfied'); t7 (s7 para s4 com 'confirm'); t8 (s7 para s3 com 'reject'). Existem também seções para 'constraints' (checkFlightPlan) e 'action' (askConfirmation).

Código 49 – XMLaw de Mitigação da Ameaça CDL09.

Ameaça: CDL10

Contexto: Cena de controle de pista

Estratégia de detecção: a estratégia de mitigação da ameaça CDL09 inseriu uma constraint chamada *checkflightPlan* que verificava quebra nos padrões de

segurança no plano de vôo. Para que a autonomia mínima também seja verificada, basta modificar a implementação desta *constraint* e verificar se o plano de vôo contempla a autonomia mínima. Desta forma, se não contemplar, o protocolo segue da forma como foi especificado, solicitando uma confirmação e assim por diante.

Ameaça: CDL11

Contexto: Todas as cenas

Estratégia de detecção: a ameaça consiste em o controlador perder o canal de comunicação e a estratégia de mitigação é solicitar ao agente ATC Façade que informe a todos os pilotos quais são os controladores disponíveis. Assim, o piloto pode se comunicar com outro controlador. A detecção de que um controlador perdeu o canal de comunicação é feita pelo agente detector de falhas. Este agente gera o evento *agent_unavailable* passando como parâmetro o controlador que não está conseguindo se comunicar. Quando este evento é detectado, ativa-se uma action que envia uma mensagem para o agente ATC Façade informado da indisponibilidade do controlador. A action diferencia entre os eventos de *agent_unavailable* através dos parâmetros. Isso permite que ela não mande mensagens ao ATC quando o que ficou indisponível foi um radar, por exemplo. Ao receber a mensagem, o ATC Façade envia uma mensagem a todos os pilotos informando os controladores disponíveis.

```
atc-law{
    // global actions
    switchStation{(failure_detector, agent_unavailable),
    br.les.SwitchStation}

    radarDown{(failure_detector, agent_unavailable),
    br.les.RadarDown}

    [ controllerDown{(failure_detector, agent_unavailable),
    br.les.ControllerDown} ]
}

// cena
groundControl {
    ...
}

// cena
take-off{
    ...
}
```



```
//cena
flight{
    ...
}

//cena
landing{
    ...
}

}//end law
```

Código 50 – XMLaw de Mitigação da Ameaça CDL11

Ameaça: CDL12

Contexto: Cena aterrissagem

Estratégia de detecção: a implementação desta estratégia ocorre de forma análoga a utilizada para a ameaça CDL09. Declara-se uma *constraint* nas transições t_3 e t_4 . Essa *constraint* verifica se o ACN da aeronave é menor ou igual ao PCN da pista. Caso não seja, a *constraint* gera o evento *constraint_not_satisfied* e evita que a transição t_3 ou t_4 dispare. Porém, o evento de *constraint_not_satisfied* faz com que as transições t_5 ou t_6 disparem. A transição t_5 muda o estado do protocolo de s_2 para s_5 . A transição t_6 muda o estado do protocolo de s_3 para s_5 . O disparo das transições t_5 ou t_6 faz com que a ação *askConfirmation* seja ativada. Esta ação envia uma mensagem para o controlador solicitando a confirmação da autorização de pouso, visto que algo fora dos padrões fora encontrado. Neste ponto o protocolo está no estado s_5 e só existem duas transições de saída. A transição t_8 dispara quando o controlador rejeita o pouso e a transição t_7 dispara quando o controlador confirma a pouso.

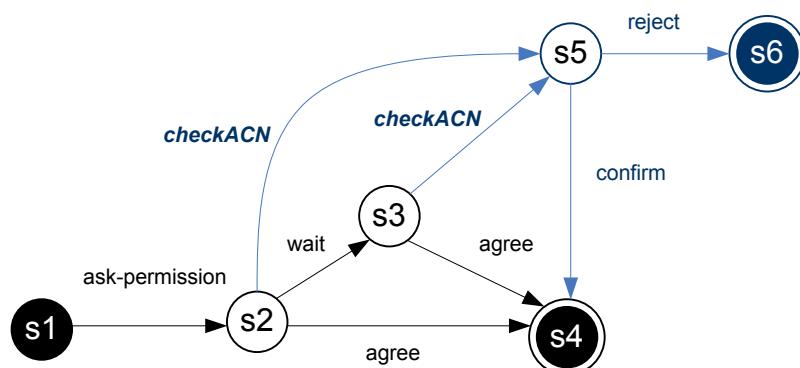


Figura 41 – Protocolo da Cena de Aterrissagem com a inclusão dos estados s5 e s6

```

landing{
    ask-permission{pilot, controller,
request(landingPermission)}
    wait{controller, pilot, inform(wait)}
    agree{controller, pilot, agree(permission)}
    | confirm{controller, pilot, confirm(accept-proposal)}
    | reject{controller, pilot, reject-proposal}
}

s1{initial}
s4{success}
s6{failure} ←

t1{s1->s2, ask-permission}
t2{s2->s3, wait}
t3{s3->s4, agree, [[checkACN]]} ←
t4{s2->s4, agree, [[checkACN]]}
t5{s2->s5, (checkACN, constraint_not_satisfied)}
t6{s3->s5, (checkACN, constraint_not_satisfied)} ←
t7{s5->s4, confirm}
t8{s5->s6, reject}

// constraints
checkACN{br.les.CheckACN} ←

// action
askConfirmation{(t5,t6), br.les. AskConfirmation} ←
}

```

Código 51 – XMLaw de Mitigação da Ameaça CDL12

Ameaça: CDL13

Contexto: Cena aterrissagem

Estratégia de detecção: para evitar que o piloto não execute o circuito de tráfego padrão ao pousar, foi adicionado ao protocolo uma etapa de monitoramento onde o piloto informa continuamente qual a sua posição e velocidade. Esta informação é transmitida através da mensagem *flightStrip*. O recebimento desta mensagem pelo mediador ativa a *action checkAndWarnCircuit*. Baseado no *flightStrip*, esta *action* verifica se o circuito está efetivamente sendo cumprido e caso não esteja avisa ao piloto e ao controlador.

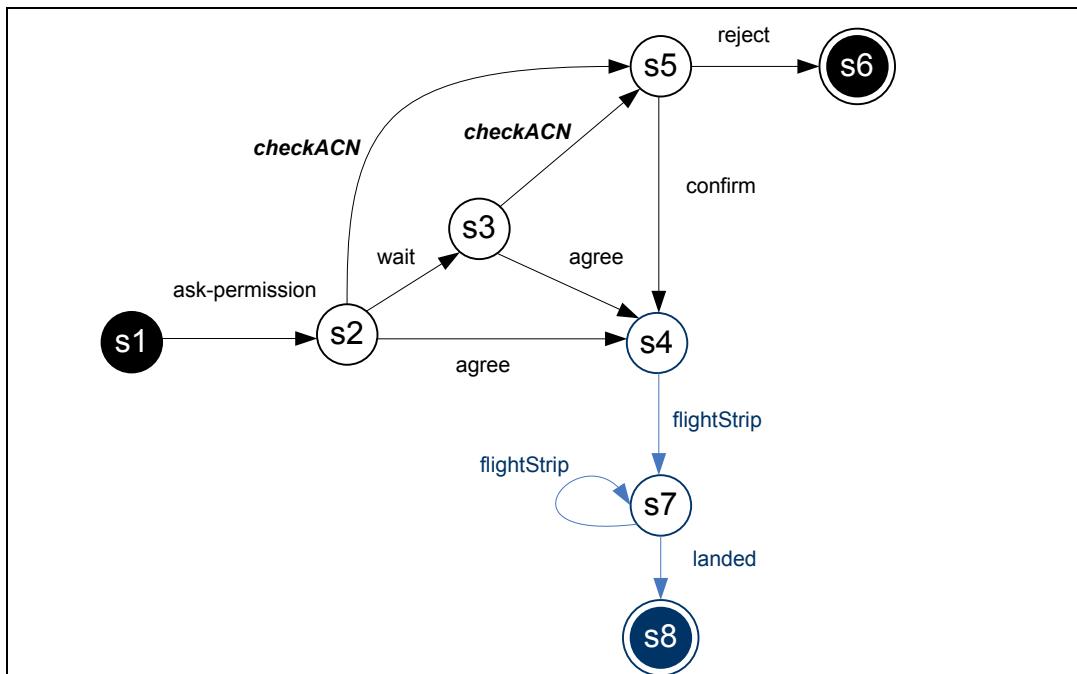


Figura 42 – Protocolo da Cena de Aterrissagem Modificado com a inclusão dos estados s7 e s8.

```

landing{
    ask-permission{pilot, controller,
    request(landingPermission)}
    wait{controller, pilot, inform(wait)}
    agree{controller, pilot, agree(permission)}
    confirm{controller, pilot, confirm(accept-proposal)}
    reject{controller, pilot, reject-proposal}
    flightStrip{pilot, controller, inform($strip)}
    landed{pilot, controller, inform(landed)}
}

s1{initial}
[s8{success}] ←
s6{failure}

t1{s1->s2, ask-permission}
t2{s2->s3, wait}
t3{s3->s4, agree, [checkACN]}
t4{s2->s4, agree, [checkACN]}
t5{s2->s5, (checkACN, constraint_not_satisfied)}
t6{s3->s5, (checkACN, constraint_not_satisfied)}
t7{s5->s4, confirm}
t8{s5->s6, reject}
t9{s4->s7, flightStrip}
t10{s7->s7, flightStrip}
t11{s7->s8, landed} ←

// constraints
checkACN{br.les.CheckACN}

// action
askConfirmation{(t5,t6), br.les. AskConfirmation}
checkAndWarnCircuit{(t9,t10),br.les.CheckAndWarnCircuit}

```

```
}
```

Código 52 – XMLaw de Mitigação da Ameaça CDL13

Ameaça: CDL14

Contexto: Cena de vôo

Estratégia de detecção: A mitigação desta ameaça pode ser implementada apenas modificando-se a estratégia de mitigação da ameaça CDL02. Para isto, basta mudar a implementação da *constraint checkAltitude* para verificar também se a aeronave está voando pelo menos 1000 pés acima do obstáculo mais alto num raio de 600 metros.

Ameaça: CDL15

Contexto: Cena de vôo

Estratégia de detecção: Mais uma vez, a implementação desta estratégia de mitigação consiste em modificar a *constraint* para verificar também se a aeronave está voando a uma altura superior a 500 pés em lugares habitados. Note-se, que embora tenha-se reutilizado a mesma *constraint* para implementar as mitigações das ameaças CDL02, CLD14 e CLD15, poderia-se, ao invés disto, ter-se criado três *constraints* diferentes, sendo uma para cada ameaça.

7.8. Dependability Explicit Computing

As especificações das leis podem tratar explicitamente conceitos de fidedignidade, e auxiliar na coleta e publicação de dados sobre fidedignidade. Estes dados podem ser utilizados, por exemplo, para auxiliar na construção de aplicações guiando decisões tanto em tempo de projeto quanto em tempo de execução.

As principais vantagens da utilização de uma abordagem de leis para a especificação de preocupações de fidedignidade são: (i) definição explícita das preocupações; (ii) coleta automática de metadados usando a infra-estrutura de

mediadores presente na maioria das abordagens de leis; e (iii) habilidade de especificar estratégias para reagir a situações não-desejadas, auxiliando na prevenção de falhas de serviço. Nesta seção, ilustra-se como a abordagem proposta nesta tese foi utilizada para implementar *Dependability Explicit Computing(DepEx)* no estudo de caso de CTA.

Em uma abordagem de DepEx é importante definir quais são os metadados de fidedignidade de maneira explícita e, também o propósito do seu uso. Neste estudo de caso, utilizou-se uma abordagem de GQM (Van Solingen and Berghout 1999) para identificar estes metadados. O raciocínio é que através da utilização do mediador e da especificação das leis é possível obter informações da execução do sistema que podem auxiliar a análise da fidedignidade do sistema. O GQM auxilia na identificação de quais informações devem ser monitoradas para alcançar os objetivos da análise. Um dos propósitos do GQM é obter formas concretas de identificar se um determinado objetivo está sendo alcançado. Isto é feito identificando-se um conjunto de objetivos, questões que auxiliam na verificação do cumprimento destes objetivos e finalmente um conjunto de métricas que respondem às questões. De acordo com o GQM, um objetivo (*goal*) deve conter em sua declaração os seguintes itens:

- Objeto – o artefato que está em estudo
- Propósito – a motivação por trás do objetivo
- Foco – o atributo do objeto em estudo
- Ponto de Vista – a perspectiva do objetivo
- Ambiente – o escopo no qual o estudo se baseia

Desta forma, utilizou-se a estrutura acima como um guia para estruturar os objetivos. Estes objetivos são apresentados a seguir:

Objetivo 01 (G01)

A equipe de qualidade de vôo deseja (ponto de vista)
analisar os pilotos (objeto)
para identificar (propósito)
o desempenho em termos de falhas cometidas (foco)
durante a operação das aeronaves em todas as fases de vôo (ambiente)

Objetivo 02 (G02)

A equipe de qualidade de vôo deseja

analisar os controladores
para identificar
o desempenho ao recomendar ações fora dos padrões estabelecidos
durante a operação das aeronaves em todas as fases de vôo

Objetivo 03 (G03)

A equipe de qualidade de vôo deseja
analisar os radares
para identificar
a confiabilidade em termos de falhas de comunicação
durante a operação das aeronaves em todas as fases de vôo

Objetivo 04 (G04)

A equipe de qualidade de vôo deseja
analisar as estações meteorológicas
para identificar
a confiabilidade em termos de falhas de comunicação
durante a operação das aeronaves em todas as fases de vôo

Objetivo 05 (G05)

A equipe de qualidade de vôo deseja
analisar o canal de comunicação do controlador
para identificar
a confiabilidade
durante a operação das aeronaves em todas as fases de vôo

A partir deste conjunto de objetivos definidos, é preciso identificar as questões que auxiliam a concluir se um determinado objetivo foi efetivamente alcançado. A estratégia seguida para identificar as questões foi reler as ameaças e entender o relacionamento dela com os objetivos. Por exemplo, a ameaça CDL01 é descrita como “aeronave se aproxima demais de outra aeronave”. Uma das causas desta aproximação indevida é por falha do piloto, que é exatamente o objetivo G01. Desta forma, a identificação desta ameaça serve como subsídio para

a coleta de dados que auxiliam na verificação do cumprimento do objetivo. A Figura 43 ilustra como o objetivo G01 pode ser verificado.

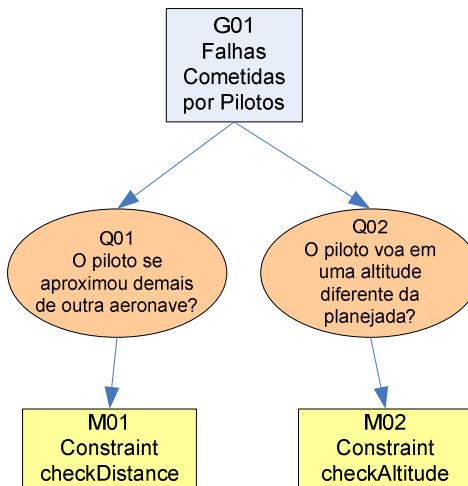


Figura 43 – GQM do Objetivo G01

As questões Q01 e Q02 são utilizadas para verificar o objetivo G01. A questão Q01 utiliza como métrica a violação da *constraint checkDistance*. Todas as vezes que esta *constraint* for violada, uma *action* é ativada como forma de atualizar o banco de dados com os metadados. O banco de dados de metadados possui a estrutura apresentada na Figura 44 e explicada na Tabela 15. Todas as vezes que a *action updateG01* é executada, ela insere no banco de dados uma tupla como exemplificado na Tabela 16.

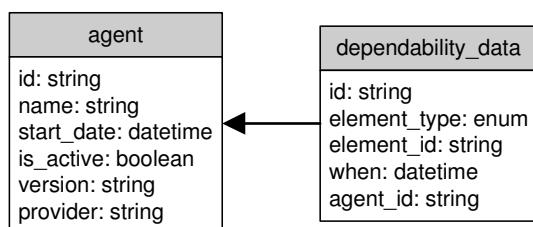


Figura 44 – Modelo do Banco de Dados

agent	dependability_data
id – identificador único do agente.	id – identificador único para o dado
name – nome do agente	element_type – tipo do elemento XMLLaw (ex: obligation, clock, ...)
start_date – data de quando o agente foi adicionado ao banco de dados	element_id – id do elemento XMLLaw
is_active – assume valor <i>true</i> se o agente está em	when – data da inserção do dado

execução	
version – versão do agente	agent_id – identificador do agente associado com estes dados
provider – organização que é responsável pelo agente	

Tabela 15 – Descrição dos Atributos do Banco de Dados

id	element_type	element_id	when	agent_id
1	constraint	checkDistance	2007-07-23 14:13:44	1

Tabela 16- Exemplo de Tupla Inserida no Banco de Dados

A partir do armazenamento destas informações, é possível responder as questões. Por exemplo, para saber se o piloto se aproximou demais da aeronave basta consultar quantas tuplas cujo campo *element_id* tem valor igual a *checkDistance*. O Código 53 mostra a cena de vôo modificada para inserir a ação cuja implementação adiciona uma tupla no banco de dados.

```

flight{    // nome da cena

    // Mensagens
    progressR{radar, controller, inform(strip,
$flightProgressStrip)}
    progressA{pilot, controller, inform(strip,
$flightProgressStrip)}
    switch{controller, pilot, inform(switch, $newController)}
    landing{pilot, controller, inform(landIntention)}

    // Estados especiais
    s1{initial}
    s3{success}

    // Transições
    t1{s1->s2, progressR, [checkDistance, checkAltitude]}
    t2{s1->s2, progressA, [checkDistance, checkAltitude]}
    t3{s2->s2, progressR, [checkDistance, checkAltitude]}
    t4{s2->s2, progressA, [checkDistance, checkAltitude]}
    t5{s2->s2, switch}
    t6{s2->s3, landing}

    // Constraints
    checkDistance{br.les.CheckDistance}
    checkAltitude{br.les.CheckAltitude}

    // Actions
    warnTcas{ ((checkDistance, constraint_not_satisfied)), 
br.les.WarnTCAS}

    warnMsaw{ ( (checkAltitude, constraint_not_satisfied) ), 
br.les.WarnMSAW}
}

```

```

warnPilot{
  (
    (checkDistance,constraint_not_satisfied),
    (checkAltitude,constraint_not_satisfied)
  ), br.les.WarnPilot}

warnController{
  (
    (checkDistance,constraint_not_satisfied),
    (checkAltitude,constraint_not_satisfied)
  ), br.les.WarnController}

warnDefroster{ (t1,t2,t3,t4), br.les.WarnDefroster}

checkInstruments{ ( t1,t2,t3,t4), br.les.CheckInstruments}
-----
| updateG01metadata{
|   (
|     (checkDistance,constraint_not_satisfied),
|     (checkAltitude,constraint_not_satisfied)
|   ), br.les.UpdateG01}
|-----
```



**Código 53 – Cena *flight* Modificada com a Inserção da *action* para
Armazenar os Dados no Banco de Dados**

Os outros objetivos são resolvidos de forma análoga ao G01. A Figura 45 mostra a aplicação do GQM para a identificação dos metadados que irão popular o banco de dados.

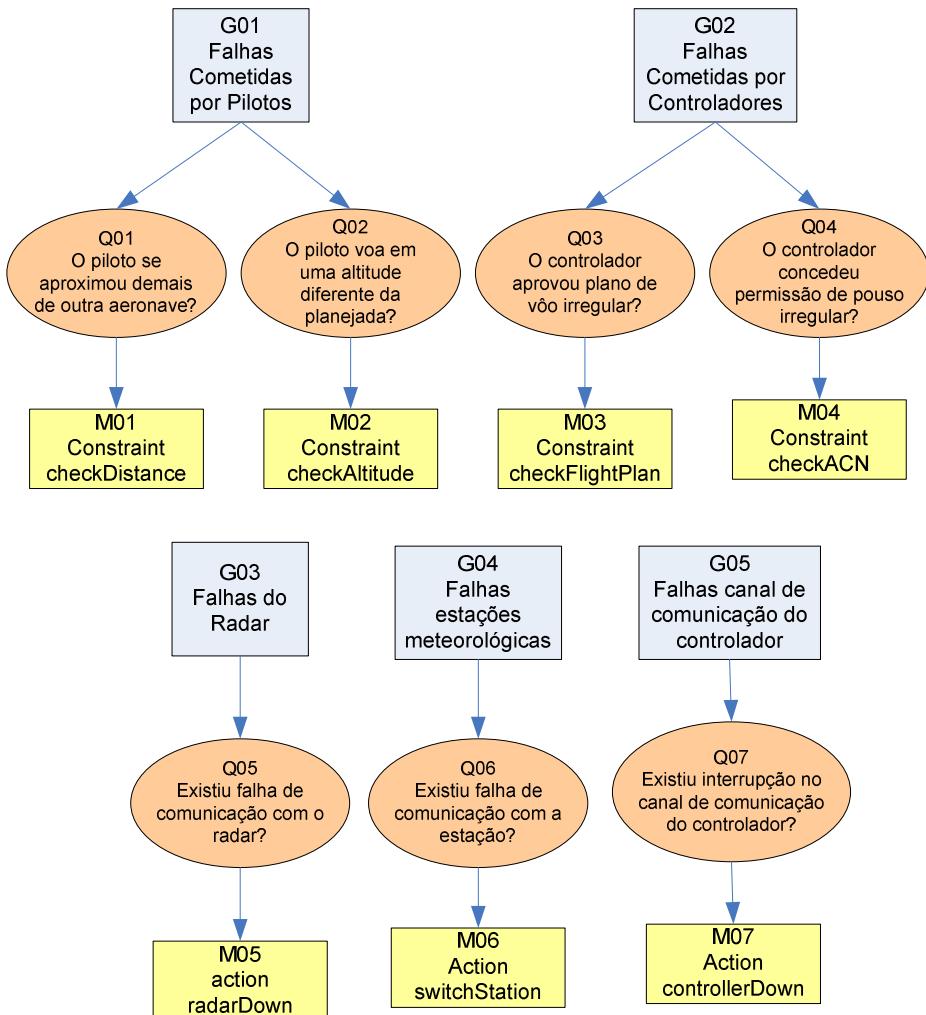


Figura 45 – GQM do Estudo de Caso para a Identificação dos Metadados

7.9.Resumo dos Artefatos de Lei Gerados no Estudo de Caso

Neste capítulo, apresentou-se como o estudo de caso foi construído. Para auxiliar a consolidar o resultado das discussões das sessões anteriores, esta seção apresenta os principais artefatos relacionados a governança gerados no estudo de caso.

7.9.1.Arquitetura

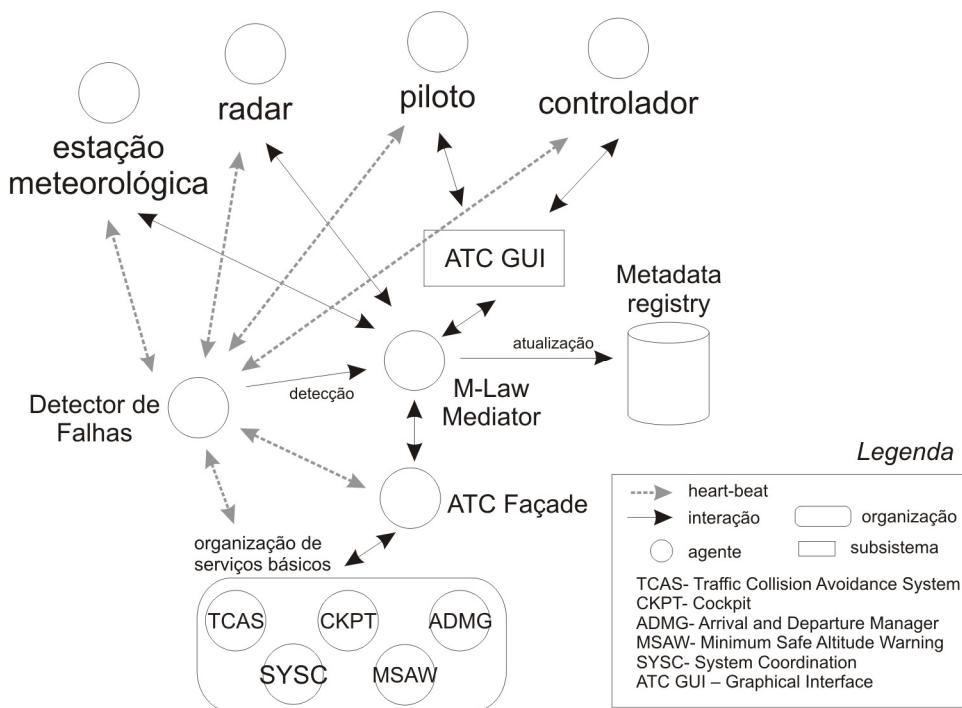


Figura 46 – Arquitetura do Estudo de Caso do Controle de Tráfego Aéreo

7.9.2.Lei

```

atc-law{
    // global actions
    switchStation{(failure_detector, agent_unavailable),
br.les.SwitchStation}

    radarDown{(failure_detector, agent_unavailable),
br.les.RadarDown}

    controllerDown{(failure_detector, agent_unavailable),
br.les.ControllerDown}

    //cena
    ######
    groundControl{ //nome da cena

    // Mensagens
    flightPlan{pilot, controller,propose(flightPlan($content)) }
    reject{controller, pilot, reject-proposal}
    accept{controller, pilot, accept-proposal}
    intent{pilot, controller, request(go-position)}
    position{controller, pilot, inform(pos, $instructions) }
    confirm{controller, pilot, confirm(accept-proposal)}

    // Estados especiais
}

```

```

s1{initial}
s3{failure}
s6{success}

// Transições
t1{s1->s2, flightPlan}
t2{s2->s3, reject}
t3{s2->s4, accept, [checkFlightPlan]}
t4{s4->s5, intent}
t5{s5->s6, position}
t6{s2->s7, (checkFlightPlan, constraint_not_satisfied) }
t7{s7->s4, confirm}
t8{s7->s3, reject}

// constraints
checkFlightPlan{br.les.CheckFlightPlan}

// action
askConfirmation{t6, br.les. AskConfirmation}

}

//cena
// #####
take-off{ //nome da cena

// Mensagens
request{pilot, controller, request(take-off)}
refuse{controller, pilot, refuse }
agree{controller, pilot, agree }
status{pilot, controller, inform($status)}
flight{pilot, controller, inform(flightPhase) }

// Estados especiais
s1{initial}
s3{failure}
s5{success}

// Transições
t1{s1->s2, request}
t2{s2->s3, refuse}
t3{s2->s4, agree}
t4{s4->s4, status}
t5{s4->s5, flight}

// Actions
warnPilot{ (t4), br.les.WarnPilot}
}

//cena
// #####
flight{ //nome da cena

// Mensagens
progressR{radar, controller, inform(strip,
$flightProgressStrip)}
progressA{pilot, controller, inform(strip,
$flightProgressStrip)}
switch{controller, pilot, inform(switch, $newController)}
landing{pilot, controller, inform(landIntention)}

```

```

// Estados especiais
s1{initial}
s3{success}

// Transições
t1{s1->s2, progressR, [checkDistance, checkAltitude] }
t2{s1->s2, progressA, [checkDistance, checkAltitude] }
t3{s2->s2, progressR, [checkDistance, checkAltitude] }
t4{s2->s2, progressA, [checkDistance, checkAltitude] }
t5{s2->s2, switch}
t6{s2->s3, landing}

// Constraints
checkDistance{br.les.CheckDistance}
checkAltitude{br.les.CheckAltitude}

// Actions
warnTcas{ ((checkDistance, constraint_not_satisfied)),
br.les.WarnTCAS}

warnMsaw{ ( (checkAltitude, constraint_not_satisfied) ),
br.les.WarnMSAW}

warnPilot{
(
    (checkDistance, constraint_not_satisfied),
    (checkAltitude, constraint_not_satisfied)
), br.les.WarnPilot}

warnController{
(
    (checkDistance, constraint_not_satisfied),
    [ (checkAltitude, constraint_not_satisfied) ]
), br.les.WarnController}

warnDefroster{ (t1,t2,t3,t4), br.les.WarnDefroster}

checkInstruments{ ( t1,t2,t3,t4), br.les.CheckInstruments}

updateG01metadata{
(
    (checkDistance, constraint_not_satisfied),
    (checkAltitude, constraint_not_satisfied)
), br.les.UpdateG01}
}

//cena
// ######
landing{
ask-permission{pilot, controller,
request(landingPermission)}
wait{controller, pilot, inform(wait)}
agree{controller, pilot, agree(permission)}
confirm{controller, pilot, confirm(accept-proposal)}
reject{controller, pilot, reject-proposal}
flightStrip{pilot, controller, inform($strip)}
landed{pilot, controller, inform(landed)}

s1{initial}
s8{success}

```

```

s6{failure}

t1{s1->s2, ask-permission}
t2{s2->s3, wait}
t3{s3->s4, agree, [checkACN]}
t4{s2->s4, agree, [checkACN]}
t5{s2->s5, (checkACN, constraint_not_satisfied)}
t6{s3->s5, (checkACN, constraint_not_satisfied)}
t7{s5->s4, confirm}
t8{s5->s6, reject}
t9{s4->s7, flightStrip}
t10{s7->s7, flightStrip}
t11{s7->s8, landed}

// constraints
checkACN{br.les.CheckACN}

// action
askConfirmation{(t5,t6), br.les. AskConfirmation}
checkAndWarnCircuit{(t9,t10), br.les.CheckAndWarnCircuit}
}

}//end law

```

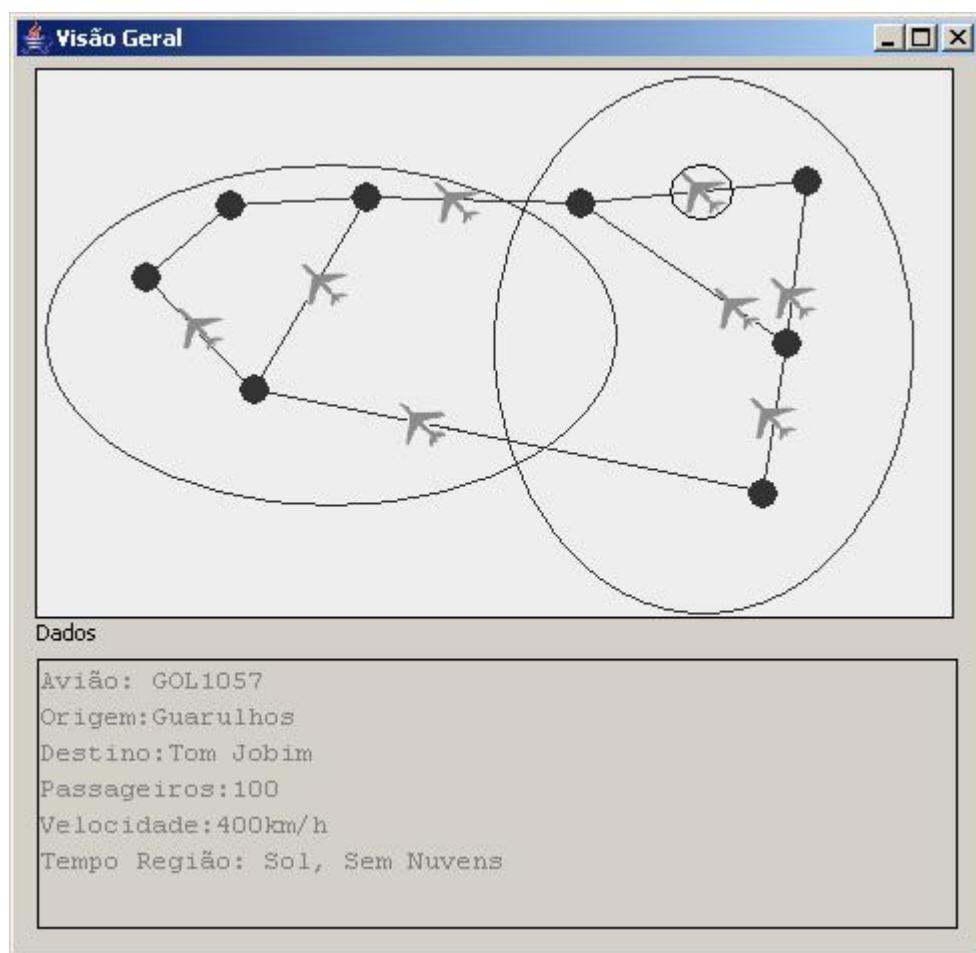


Figura 47 – Tela da Visão Geral do Sistema

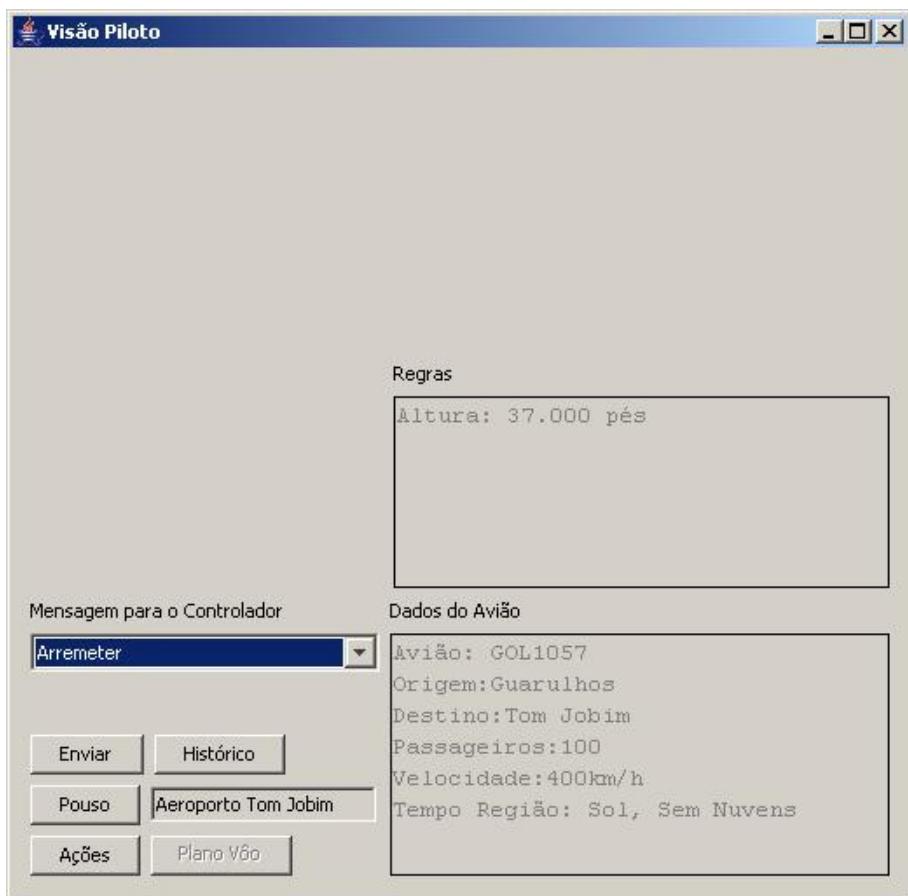


Figura 48 – Tela Representando o Agente Piloto

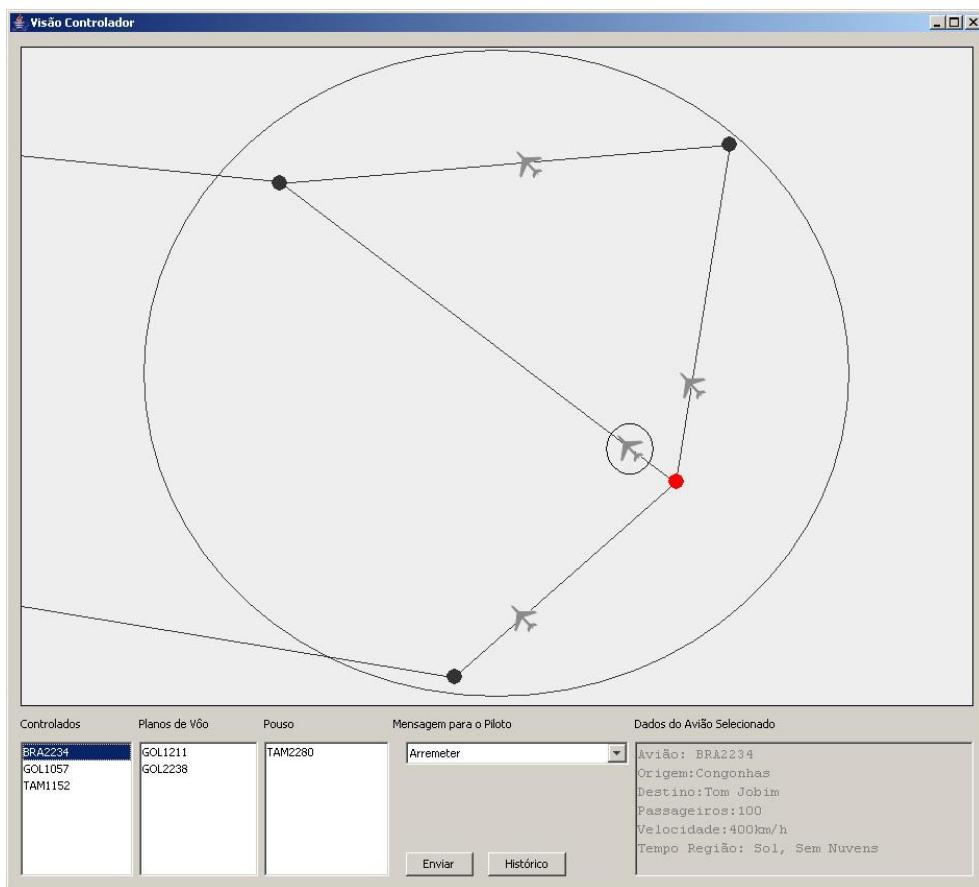


Figura 49 – Tela Representando a Visão do Agente Controlador

7.10.Considerações Finais

O estudo de caso foi implementada utilizando o XMLaw e o M-Law e os agentes foram implementados em Java. O estudo de caso de CTA possui elementos de incerteza, criticalidade, distribuição e complexidade de interações que certamente o tornam adequado para a experimentação da abordagem proposta. Através do XMLaw foi possível representar vários aspectos de governança. Mas o principal aspecto a ser destacado é que a utilização de uma abordagem de governança pode efetivamente incorporar aspectos de fidedignidade. Primeiramente, o domínio do problema foi apresentado e vários casos de uso foram definidos. Uma arquitetura baseada em agentes foi proposta para representar os atores que fazem parte do sistema. A partir daí, foi realizada uma análise detalhada de várias ameaças que podem impactar o bom funcionamento do sistema. Estas ameaças foram todas mitigadas através da utilização dos mecanismos propostos nesta tese.

O modelo conceitual do XMLaw permitiu expressar as situações onde se verifica a existência de uma ameaça. Além disso, também foi possível expressar as estratégias de mitigação utilizando a própria linguagem.

Foram simuladas situações em que a ameaça se tornava concreta. O middleware M-Law foi capaz de perceber estas situações, através da especificação das leis e do monitoramento da execução do sistema, e agir de acordo com as estratégias de implementação especificada.

De forma geral, pode-se concluir que a abordagem proposta nesta tese foi capaz de representar estratégias de mitigação apresentadas e o middleware foi capaz de monitorar e agir de acordo com as especificações. Desta forma, mostra-se que preocupações de fidedignidade puderam ser tratadas com a abordagem proposta.

8 Conclusões

As abordagens baseadas em leis vêm ganhando uma importância crescente na comunidade de sistemas multi-agentes. Elas fornecem abstrações que permitem a modelagem e a decomposição do problema que são adequadas a domínios de problemas onde existe a necessidade de impor algum grau de controle sobre a autonomia dos agentes.

Por outro lado, a sociedade depende cada vez mais de sistemas que estão conectados entre si e que precisam se comportar de acordo com regras muito bem definidas. Pode-se considerar que cada um destes subsistemas é um agente de software que interage com os outros agentes para prover as várias funcionalidades dos sistemas. A medida que a dependência aumenta, aumenta também a necessidade de que estes sistemas sejam construídos de tal forma que possamos confiar que eles irão funcionar com os níveis especificados de segurança, confiabilidade e desempenho, dentre outros atributos.

Nesta tese, apresenta-se a idéia de utilizar leis de interação para especificar, monitorar e eventualmente implementar soluções relacionadas com fidedignidade. Ou seja, incorpora-se em uma abordagem de governança uma nova dimensão ou aspecto que ainda não havia sido explorada. Esta união é adequada uma vez que se reutiliza as abstrações fornecidas nas abordagens de leis e que permitem a especificação de várias situações complexas encontradas em sistemas interativos. A incorporação dos aspectos de fidedignidade foi realizada utilizando os próprios elementos de lei do modelo conceitual. Além do aspecto relacionado a expressividade, a grande maioria das abordagens de leis impõe alguma forma de controle, seja através de mediadores ou auto-regulamentação (Chopinaud, El Fallah Seghrouchni et al. 2006). Esta estrutura, também pode ser reutilizada para monitorar e coletar informações durante a execução do sistema com fins de fidedignidade.

O principal problema abordado neste trabalho foi a falta de garantia 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, foram propostos instrumentos de governança que os projetistas especifiquem e implementem sistemas multi-agentes abertos levando em consideração com preocupações de fidedignidade. Estes mecanismos consideram que os detalhes da arquitetura e a implementação dos agentes são inacessíveis; (ii) os agentes interagem através de troca de mensagens; (iii) é possível especificar quais são as interações e comportamentos esperados do sistema a priori.

A definição tradicional de falha encontrada na literatura e adotada neste trabalho considera uma falha como o fato do sistema não apresentar o comportamento esperado de acordo com a sua especificação. Entretanto, em sistemas multi-agentes, os agentes podem agir deliberadamente para descumprir uma determinada especificação, uma obrigação por exemplo. Desta forma, uma questão em aberto é como considerar esta situação. Deve-se considerar isto uma falha ou encontrar uma nova maneira de lidar com este ato deliberado? A resposta a esta pergunta possui uma relação direta com a abordagem proposta, visto que atualmente, o XMLaw considera este ato deliberado como uma falha.

8.1.Resultados

O principal resultado obtido com esta tese foi a experimentação com a incorporação da dimensão de fidedignidade em uma abordagem de leis. Para atingir este resultado, vários resultados intermediários também foram produzidos, dentre os quais é possível destacar:

- Revisão e especificação do modelo conceitual – o modelo conceitual do XMLaw foi criado pela primeira vez nos trabalhos realizados em (Paes 2005; Paes, Carvalho et al. 2005). Nesta tese, a maioria dos elementos originais foram revistos tanto em termos de estrutura quanto em termos do ciclo de vida. Além disso, novos eventos foram adicionados (*agent_unavailable*, *message_compliant* ...). Também é contribuição desta tese o detalhamento do modelo de eventos, além da estrutura de descrição dos elementos do modelo.
- Nova linguagem para especificação das leis – o modelo conceitual era representado em uma linguagem baseada em XML. Embora

ainda seja possível utilizar esta linguagem, foi proposta uma nova linguagem de representação. A nova linguagem permite uma especificação de lei de forma mais compacta.

- Comparação detalhada do modelo conceitual em relação as outras abordagens – foi realizado um estudo comparativo detalhado do modelo conceitual do XMLaw com as abordagens LGI e Instituições Eletrônicas. Este estudo permite, por exemplo, auxiliar na tomada de decisão sobre qual tecnologia escolher no momento de implementar as leis. Além disso, mostrou-se a linha de raciocínio de como uma lei implementada em XMLaw pode ser mapeada para LGI.
- Experimentação através de estudos de caso – ao todo esta tese apresenta cinco estudos de caso, sendo quatro deles pequenos e um mais detalhado. A implementação dos estudos de caso podem ser consideradas como um dos resultados desta tese. Além disso, a descrição do estudo de caso apresentado na Seção 7, pode ser utilizado também por outras abordagens de implementação de leis.
- Publicações – os resultados parciais gerados durante o desenvolvimento deste trabalho foram publicados em uma série de eventos nacionais e internacionais, dentre eles alguns eventos classificados como ‘A’ pela CAPES. Segue a lista de publicações: (Carvalho, Paes et al. 2004; Machado, Carvalho et al. 2004; Paes, Almeida et al. 2004; Paes, Carvalho et al. 2004; Carvalho, Paes et al. 2005; Carvalho, Paes et al. 2005; Carvalho, Paes et al. 2005; Felicíssimo, Lucena et al. 2005; Paes 2005; Paes, Carvalho et al. 2005; Paes, Lucena et al. 2005; Rodrigues, Carvalho et al. 2005; Carvalho, Almeida et al. 2006; Carvalho, Brandão et al. 2006; Carvalho, Lucena et al. 2006; Gatti, Carvalho et al. 2006; Gatti, Carvalho et al. 2006; Gatti, Paes et al. 2006; Paes, Gatti et al. 2006; Carvalho 2007; Carvalho, Choren et al. 2007; Paes, Carvalho et al. 2007; Paes, Lucena et al. 2007; Paes, Lucena et al. 2007)

8.2.Limitações

O trabalho proposto possui as seguintes limitações que poderiam ser abordadas em trabalhos futuros:

- O monitoramento e *enforcement* realizado pelo M-Law é feito de forma centralizada – esta característica introduz um único ponto de falha no sistema e pode ser um ponto de gargalo no sistema em termos de escalabilidade. Várias soluções poderiam ser dadas a este problema tal como a adoção do *enforcement* totalmente descentralizado como feito em (Minsky 2005) ou híbrida como proposto em (Ansari 2006). Entretanto cada uma possui vantagens e desvantagens. Em relação a descentralização, por exemplo, não seria possível especificar leis de um ponto de vista global sem a introdução de um mecanismo de sincronização entre os estados dos mediadores descentralizados. Em termos de escalabilidade, uma solução seria manter o *enforcement* centralizado, mas utilizar um *pool* de mediadores que funcionariam como平衡amento de carga.
- Ausência de um modelo formal do modelo conceitual – um modelo formal do modelo conceitual permitiria a realização de verificações estruturais e até mesmo semânticas da especificação de uma lei (Carvalho, Brandão et al. 2006). Um complicador para a geração deste modelo formal de todos os elementos do XMLaw são os elementos que envolvem código externo, como a Action e a Constraint. A dificuldade é que estes elementos possuem a semântica dependente do código Java que os implementa.

8.3.Trabalhos Futuros

Durante a evolução deste trabalho, foram levantadas várias discussões sobre direcionamento de trabalhos futuros. Nesta seção, resumem-se algumas destes direcionamentos com o intuito de documentar linhas de evolução que certamente agregariam valor à área de pesquisa desta tese.

8.3.1.Agentes Deliberativos Normativos

Uma importante característica dos agentes é que eles podem reagir a ambientes mudam freqüentemente. Entretanto, se os protocolos que os agentes usam para reagir às mudanças no ambiente são fixos, os agentes não têm como reagir a mudanças não previstas. Por exemplo, se um agente percebe que outro agente está trapaceando, ele não conseguiria mudar as suas reações para se proteger. Em algumas situações é adequado permitir a violação inteligente das leis. Ou seja, os agentes são capazes de automaticamente interpretar as leis e identificar, de acordo com o contexto, qual a ação ele deve executar (Castelfranchi, Dignum et al. 1999; Boella and van der Torre 2003; Kollingbaum and Norman 2003).

8.3.2.Verificação Formal

Na medida em que a complexidade da lei aumenta, aumentam também as chances de se especificar uma lei de forma incorreta. A verificação formal das leis permitirá, em tempo de especificação, a identificação de inconsistências nas leis. Além disso, a especificação formal também pode ser utilizada como base de um mecanismo de *reasoning* de agentes deliberativos normativos. Um exemplo de formalização de leis pode ser visto em (Esteva, Rodriguez et al. 2001).

8.3.3.Geração de Código dos Agentes

As leis descrevem o comportamento esperado dos agentes sob o ponto de vista de interação. Assim, é possível a partir da especificação das leis, gerar parte do código do agente que irá interagir. O código gerado deverá conter métodos parcialmente implementados que serão preenchidos com as particularidades de implementação de cada agente. A geração automática diminuiria os erros de codificação dos agentes e aceleraria o tempo de desenvolvimento de novos agentes.

8.3.4.Introdução Explícita de Elementos de Fidedignidade

Nesta tese, as preocupações de fidedignidade foram tratadas com as abstrações disponíveis no modelo de leis. Entretanto, pode-se pensar em tornar mais explícita a noção de fidedignidade através da introdução de novos elementos no modelo conceitual. Por exemplo, ao se falar de *backward recovery* poderia-se introduzir elementos tais como *checkpoints* e *recovery blocks* (Randell and Xu

1995). O mesmo poderia ser aplicado para o gerenciamento de exceções, por exemplo, alguns eventos poderiam ser do tipo exceção, poderiam existir *actions* identificadas explicitamente como *exception handlers*, ou ainda uma definição explícita de contextos para as exceções.

8.3.5.Mediadores Distribuídos

A implementação atual do mediador do XMLaw utiliza um mediador centralizado. Em muitos sistemas, esta centralização pode não ser adequada. Existem várias alternativas para descentralizar a mediação, dentre elas a utilização de um conjunto de mediadores que balanceiam a carga ou a mediação totalmente descentralizada conforme proposto em (Minsky and Ungureanu 2000). Em relação ao XMLaw, foi realizado um estudo inicial (Ansari 2006), mas o trabalho não foi totalmente concluído.

8.3.6.Integração com Tecnologias de Arquitetura Orientada a Serviços

Nota-se uma forte tendência das grandes corporações na adoção de estratégias de gerenciamento da área de TI em função dos serviços que ela fornece. Dentre estas práticas, existem práticas gerenciais (Commerce 2007), arquiteturas (Papazoglou and Georgakopoulos 2003) e tecnologias de implementação (Erl 2004). A idéia de se utilizar as abstrações de agentes para representar os serviços já foi proposta anteriormente em (Singh and Huhns 2005). Logo, como um forma de transferência de tecnologia, as abordagens de leis poderiam utilizar as tecnologias de implementação de arquitetura orientada a serviços, tais como os Webservices.

8.3.7.Condução de Experimentos Através de Mutantes

Os agentes utilizados para a implementação dos estudos de caso desta tese foram programados para se comportar exatamente da forma desejada. Ou seja, em algumas situações desejava-se que eles exibissem um comportamento correto e assim o agente era programado, e em outras situações desejava-se que eles descumprisem as leis. Entretanto, em um sistema real, muitas vezes o comportamento do sistema como um todo emerge de uma seqüência não prevista de interações. Desta forma, seria bastante útil estudar como aplicar mutantes na construção de agentes para simular comportamentos não previstos a priori. A

técnica de mutantes fornece, basicamente, uma forma sistemática de geração de casos de teste e de avaliação de quão adequado é um conjunto de testes (Vincenzi, Simão et al. 2006).

Referências

- Almeida, H., Â. Perkusich, et al. (2006). A Component Model to Support Dynamic Unanticipated Software Evolution. International Conference on Software Engineering and Knowledge Engineering (SEKE'06), San Francisco, USA.
- Ansari, S. (2006). Decentralized mediator in an Open Multi-Agent System. Rapport de stage. Nantes, Ecole Polytechnique de l'Université de Nantes.
- Ashri, R., T. R. Payne, et al. (2006). "Using Electronic Institutions to secure Grid environments."
- Avizienis, A., J. C. Laprie, et al. (2004). Dependability and its threats: a taxonomy. Building the information society, 18th IFIP World Computer Congress, Toulouse, France.
- Avizienis, A., J. C. Laprie, et al. (2004). "Basic concepts and taxonomy of dependable and secure computing." Dependable and Secure Computing, IEEE Transactions on **1**(1): 11-33.
- Batista, T. and N. Rodriguez (2000). "Dynamic reconfiguration of component-based applications." Software Engineering for Parallel and Distributed Systems, 2000. Proceedings. International Symposium on: 32-39.
- Bellifemine, F., A. Poggi, et al. (1999). "JADE—A FIPA-compliant agent framework." Proceedings of PAAM **99**: 97–108.
- Boehm, B. W. (1981). Software Engineering Economics, Prentice Hall PTR Upper Saddle River, NJ, USA.
- Boehm, B. W. and V. R. Basili (2001). "Software Defect Reduction Top 10 List." IEEE Computer **34**(1): 135-137.
- Boella, G. and L. van der Torre (2003). "Attributing mental attitudes to normative systems." Proceedings of the second international joint conference on Autonomous agents and multiagent systems: 942-943.
- Boissier, O., J. F. Hübner, et al. (2006). Organization Oriented Programming: From Closed to Open Organizations. Engineering Societies in the Agents World VII 7th International Workshop, ESAW 2006 Dublin, Ireland, Revised Selected and Invited Papers Series: Lecture Notes in Computer Science **4457**.
- Bou, E., M. López-Sánchez, et al. (2006). "Norm Adaptation of Autonomic Electronic Institutions with Multiple Goals." ITSSA journal International Transactions on Systems Science and Applications **1**(3): 227--238.
- Box, D., D. Ehnebuske, et al. (2000). Simple Object Access Protocol (SOAP) 1.1, May.

- Carvalho, G. R. d. (2007). G-Frameworks: Uma Abordagem para a Reutilização de Leis de Interação em Sistemas Multiagentes Abertos. Departamento de Informática. Rio de Janeiro, PUC-Rio. **Phd:** 186.
- Carvalho, G. R. d., H. Almeida, et al. (2006). Dynamic Law Evolution in Governance Mechanisms for Open Multi-Agent Systems. Second Workshop on Software Engineering for Agent-oriented Systems (SEAS 2006). Florianópolis, Brasil.
- Carvalho, G. R. d., A. Brandão, et al. (2006). Interaction Laws Verification Using Knowledge-based Reasoning. Workshop on Agent-Oriented Information Systems (AOIS-2006). Hakodate, Japan.
- Carvalho, G. R. d., R. Choren, et al. (2007). Uma Abordagem para o Reuso de Leis de Interacão em Sistemas Multi-Agentes. Simpósio Brasileiro de Engenharia de Software. João Pessoa, Brasil.
- Carvalho, G. R. d., C. J. P. d. Lucena, et al. (2006). Refinement Operators to Facilitate the Reuse of Interaction Laws in Open Multi-Agent Systems. 5th International Workshop on Software Engineering for Large-scale Multi-Agent Systems (SELMAS). Shanghai, China.
- Carvalho, G. R. d., R. d. B. Paes, et al. (2005). Increasing Software Infrastructure Dependability through a Law Enforcement Approach. International Symposium on Normative Multiagent Systems (NORMAS2005). Hatfield, England.
- Carvalho, G. R. d., R. d. B. Paes, et al. (2004). Towards a Risk Driven Method for Developing Law Enforcement Middleware. Third International Workshop on Agent-Oriented Methodologies. Vancouver, Canada.
- Carvalho, G. R. d., R. d. B. Paes, et al. (2005). Extensions on Interaction Laws in Open Multi-Agent Systems. Software Engineering for Agent-oriented Systems (SEAS 05). Uberlândia, Brazil.
- Carvalho, G. R. d., R. d. B. Paes, et al. (2005). Governing the Interactions of an Agent-based Open Supply Chain Management System. Monografias em Ciência da Computação nº 29/05, Departamento de Informática, PUC-Rio.
- Castelfranchi, C., F. Dignum, et al. (1999). "Deliberative normative agents: Principles and architecture." Proceedings of ATAL: 364–378.
- Charette, R. N. (1989). Software engineering risk analysis and management, McGraw-Hill, Inc. New York, NY, USA.
- Chen, Y., P. Li, et al. (2005). Web Services Dependability and Performance Monitoring. 21st UK Performance Engineering Workshop, United Kingdom.
- Chopinaud, C., A. El Fallah Seghrouchni, et al. (2006). Dynamic Self-control of autonomous agents. Programming Multiagents Systems, Springer-Verlag: 41--57.
- Clocksin, W. F. and C. S. Mellish (1984). Programming in Prolog, Springer-Verlag New York, Inc. New York, NY, USA.
- Commerce, O. o. G. (2007). Service Design, The Stationery Office.

- Coutinho, L. R., J. S. Sichman, et al. (2005). Modeling Organization in MAS: A Comparison of Models. First Workshop on Software Engineering for Agent-oriented Systems - SEAS 2005. C. J. P. d. Lucena, M. Blois, R. Choren and V. T. d. Silva. Uberlândia - Brasil.
- Cristian, F. (1991). "Understanding fault-tolerant distributed systems." Commun. ACM **34**(2): 56-78.
- Cugola, G., E. Di Nitto, et al. (1998). "Exploiting an event-based infrastructure to develop complex distributed systems." Proceedings of the 20th International Conference on Software Engineering (ICSE'98): 261–270.
- Cuni, G., M. Esteva, et al. (2004). "MASFIT: Multi-Agent System for FIsh Trading." ECAI **16**: 710.
- DeLoach, S. A. and E. Matson (2004). An Organizational Model for Designing Adaptive Multiagent Systems. The AAAI-04 Workshop on Agent Organizations: Theory and Practice.
- Dignum, F. (2002). "Abstract norms and electronic institutions." Proceedings of the International Workshop on Regulated Agent-Based Social Systems: Theories and Applications (RASTA'02), Bologna: 93–104.
- Dignum, V. and F. Dignum (2001). 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.
- Dignum, V., J. Vazquez-Salceda, et al. (2004). "A model of almost everything: norms, structure and ontologies in agent organizations." Autonomous Agents and Multiagent Systems, 2004. AAMAS 2004. Proceedings of the Third International Joint Conference on: 1498-1499.
- Erl, T. (2004). Service-oriented architecture, Prentice Hall.
- Esteva, M. (2003). Electronic Institutions: from specification to development. l'Institut d'Investigació en Intel·ligència Artificial. Barcelona, Universitat Autònoma de Barcelona. **19**.
- Esteva, M., J. A. Rodriguez, et al. (2001). "On the formal specifications of electronic institutions." Agent-mediated Electronic Commerce (The European AgentLink Perspective) **1191**: 126–147.
- Esteva, M., B. Rosell, et al. (2004). "AMELI: An Agent-Based Middleware for Electronic Institutions." Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 1: 236-243.
- Fayad, M., D. Schmidt, et al. (1999). Building application frameworks: object-oriented foundations of framework design, John Wiley & Sons.
- Felicíssimo, C., C. J. P. d. Lucena, et al. (2005). Normative Ontologies to Define Regulations Over Roles in Open Multi-Agent Systems. AAAI Fall Symposium on Roles- an interdisciplinary perspective. Hyatt Crystal City in Arlington, Virginia.

- Fipa, A. C. L. (2002). "Message Structure Specification." World Wide Web, <http://www.fipa.org/specs/fipa00061>.
- Gamma, E., R. Helm, et al. (1995). Design patterns: elements of reusable object-oriented software, Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.
- Garcia-Camino, A., P. Noriega, et al. (2005). "Implementing norms in electronic institutions." Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems: 667-673.
- Gärtner, F. C. (1999). "Fundamentals of fault-tolerant distributed computing in asynchronous environments." ACM Computing Surveys **31**(1): 1-26.
- Gatti, M. A. d. C., G. R. d. Carvalho, et al. (2006). Structuring a Law Case for Law-Governed Open Multi-Agent Systems. Monografias em Ciência da Computação. Rio de Janeiro, PUC-Rio. **27/06**.
- Gatti, M. A. d. C., G. R. d. Carvalho, et al. (2006). O Rationale da Fidedignidade em Sistemas Multiagentes Abertos Governados por Leis. Second Workshop on Software Engineering for Agent-oriented Systems (SEAS 2006). Florianópolis, Brasil.
- Gatti, M. A. d. C., C. J. P. d. Lucena, et al. (2006). 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.
- Gatti, M. A. d. C., R. d. B. Paes, et al. (2006). 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.
- Group, O. M. (2007). " UML 2.1.1." Retrieved 22 de Agosto de 2007, from <http://www.uml.org/>.
- Guttmann, R. H., A. G. Moukas, et al. (1998). "Agent-mediated electronic commerce: a survey." Knowl. Eng. Rev. **13**(2): 147-159.
- Hannoun, M., O. Boissier, et al. (2000). 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.
- Hannoun, M., O. Boissier, et al. (2000). 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.
- Hewitt, C. (1986). "Offices are open systems." ACM Trans. Inf. Syst. **4**(3): 271-287.
- Hübner, J. F. (2003). Um Modelo de Reorganização de Sistemas Multiagentes. Escola Politécnica. São Paulo, Universidade de São Paulo. **PhD**: 178.
- Hübner, J. F., J. S. Sichman, et al. (2006). S-moise+: A middleware for developing organised multi-agent systems. Coordination, Organizations,

- Institutions, and Norms in Multi-Agent Systems, Springer-Verlag Lecture Notes in AI. **3913**: 64–78.
- Inc., D. L. (2007). "The Challenge of Complexity, accessed in January, 2007." from <http://domaindrivendesign.org/>.
- Kaâniche, M., J. C. Laprie, et al. (2000). "A Dependability-Explicit Model for the Development of Computing Systems." Proc. SAFECOMP 2000: 107-116.
- Kan, S. H. (2002). Metrics and Models in Software Quality Engineering, Addison-Wesley Professional.
- Karolak, D. W. (1996). Software engineering risk management, IEEE Computer Society Press Los Alamitos, Calif.
- Kollingbaum, M. J. and T. J. Norman (2003). "NoA-A Normative Agent Architecture." Eighteenth International Joint Conference on Artificial Intelligence IJCAI 3: 1465–1466.
- Krey, N. C. (2006). 2005 NALL REPORT - Accident Trends and Factors for 2004. K. Hummel, K. D. Murphy and D. Wright, AOPA Air Safety Foundation.
- Lapinsky, S. E., J. Weshler, et al. (2004). "Handheld computers in critical care." feedback.
- Laprie, J.-C. and B. Randell (2004). "Basic Concepts and Taxonomy of Dependable and Secure Computing." IEEE Trans. Dependable Secur. Comput. **1**(1): 11-33.
- Machado, R. P., G. R. d. Carvalho, et al. (2004). Applying Ontologies in Open Mobile Systems. OOPSLA'04 Workshop on Building Software for Pervasive Computing. Vancouver, Canada.
- Mackinnon, T., S. Freeman, et al. (2001). "Endo-Testing: Unit Testing with Mock Objects." Extreme Programming Examined: 287-301.
- Mahmood, N., W. David, et al. (2005). "A maturity model for the implementation of software process improvement: an empirical study." J. Syst. Softw. **74**(2): 155-172.
- Matthews, S. (2002). Future developments and challenges in aviation safety. Future developments and challenges in aviation safety, Flight Safety Foundation. **21**: 1--12.
- Meier, R. and V. Cahill (2005). "Taxonomy of Distributed Event-Based Programming Systems." The Computer Journal **48**(5): 602.
- Menezes, P. B. (1997). Linguagens Formais e Autômatos. Porto Alegre.
- Meyer, B. (2003). "The Power of Abstraction, Reuse and Simplicity: An Object-Oriented Library for Event-Driven Design." Festschrift in Honor of Ole-Johan Dahl.
- Michael, B., S. Alberto, et al. (2006). "CMieux: adaptive strategies for competitive supply chain trading." SIGecom Exch. **6**(1): 1-10.
- Minsky, N. H. (2003). "On conditions for self-healing in distributed software systems." Autonomic Computing Workshop, 2003: 86-92.

- Minsky, N. H. (2005). Law Governed Interaction (LGI): A Distributed Coordination and Control Mechanism - (An Introduction, and a Reference Manual), Rutgers University.
- Minsky, N. H. (2005). "On a principle underlying self-healing in heterogeneous software." Journal of Integrated Computer-Aided Engineering.
- Minsky, N. H. and T. Murata (2004). "On manageability and robustness of open multi-agent systems." Computer Security, Dependability, and Assurance. lncs 2940.
- Minsky, N. H. and D. Rozenshtein (1987). "A law-based approach to object-oriented programming." Conference on Object Oriented Programming Systems Languages and Applications: 482-493.
- Minsky, N. H. and V. Ungureanu (2000). "Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems." ACM Trans. Softw. Eng. Methodol. **9**(3): 273-305.
- Murata, T. and N. H. Minsky (2003). "On Monitoring and Steering in Large-Scale Multi-Agent Systems." 2 ndInternational Workshop on Software Engineering for Large-Scale Multi-Agent Systems (ICSE-SELMAS 2003), Portland Or, USA, May.
- Ndovie, B. (1994). Simulation of a conflict management system for air traffic control. Second International Working Conference on CKBS. DAKE Centre, University of Keele.
- Noriega, P. (1997). Agent mediated auctions: The Fishmarket Metaphor, Universitat Autònoma de Barcelona.
- Nooshin, A. (2003). "The impact of software process improvement on quality: in theory and practice." Inf. Manage. **40**(7): 677-690.
- Paes, R. d. B. (2005). Regulando a Interação de Agentes em Sistemas Abertos - uma Abordagem de Leis. Informática. Rio de Janeiro, PUC-Rio. **Master:** 119.
- Paes, R. d. B., H. Almeida, et al. (2004). Enforcing Interaction Protocols in Multi-Agent Systems. Monografias em Ciência da Computação. Rio de Janeiro, PUC-Rio. **09/04**.
- Paes, R. d. B., G. R. d. Carvalho, et al. (2004). A conceptual architecture for law-governed open multi-agent systems. Argentine Symposium on Software Engineering. Córdoba - Argentina.
- Paes, R. d. B., G. R. d. Carvalho, et al. (2007). Enhancing the Environment with a Law-Governed Service for Monitoring and Enforcing Behavior in Open Multi-Agent Systems. Environments for Multi-Agent Systems III, Third International Workshop, E4MAS 2006, Hakodate, Japan, May 8, 2006, Selected Revised and Invited Papers. D. Weyns, H. V. D. Parunak and F. Michel, Springer. **4389**: 221--238.
- Paes, R. d. B., G. R. d. Carvalho, et al. (2005). Specifying Laws in Open Multi-Agent Systems. Agents, Norms and Institutions for Regulated Multiagent Systems (ANIREM). Utrecht, The Netherlands.

- Paes, R. d. B., M. A. d. C. Gatti, et al. (2006). A Middleware for Governance in Open Multi-Agent Systems. Monografias em Ciência da Computação. Rio de Janeiro, PUC-Rio. **33/06**.
- Paes, R. d. B., C. J. P. d. Lucena, et al. (2005). Governing Agent Interaction in Open Multi-Agent Systems. Monografias em Ciência da Computação. Rio de Janeiro, PUC-Rio. **30/05**.
- Paes, R. d. B., C. J. P. d. Lucena, et al. (2007). Incorporation of Dependability Concerns in the Specification of Multi-Agent Interactions by Using a Law Approach. Monografias em Ciência da Computação. PUC-Rio. Rio de Janeiro, Pontifícia Universidade Católica do Rio de Janeiro.
- Paes, R. d. B., C. J. P. d. Lucena, et al. (2007). Using Interaction Laws to Implement Dependability Explicit Computing in Open Multi-Agent Systems. Simpósio Brasileiro de Engenharia de Software (SBES). João Pessoa, Brasil.
- Papazoglou, M. P. and D. Georgakopoulos (2003). "Service-Oriented Computing." Communications of the ACM **46**(10): 25-28.
- Rahman, H. A., K. Beznosov, et al. (2006). Identification of Sources of Failures and their Propagation in Critical Infrastructures from 12 Years of Public Failure Reports. Third International Conference on Critical Infrastructures. Alexandria, VA.
- Randell, B. and J. Xu (1995). "The evolution of the recovery block concept." Software Fault Tolerance: 1-22.
- Ripper, P. S., M. F. Fontoura, et al. (2000). "V-Market: A framework for agent e-commerce systems." World Wide Web **3**(1): 43--52.
- Rocher, G. (2006). The Definitive Guide to Grails (Definitive Guide), APRESS.
- Rodrigues, L. F. C., G. R. d. Carvalho, et al. (2005). Towards an Integration Test Architecture for Open MAS. Software Engineering for Agent-oriented Systems (SEAS 05). Uberlândia, Brazil.
- Rodriguez-Aguilar, J. A. (2001). On the Design and Construction of Agent-mediated Electronic Institutions. IIIA. Phd.
- Rogério, C. (2007). "Regulamento de Tráfego Aéreo." Retrieved June, 11th, 2007, from http://www.airandinas.com/sala_regulamento.html.
- Serugendo, G., J. Fitzgerald, et al. (2006). Dependable Self-Organising Software Architectures - An Approach for Self-Managing Systems. London, School of Computer Science and Information Systems, Birkbeck College. **BBKCS-06-05**.
- Silva, V. T. d. (2004). Uma Linguagem de Modelagem para Sistemas Multi-agentes Baseada em um Framework Conceitual para Agentes e Objetos. Departamento de Informática - Laboratório de Engenharia de Software. Rio de Janeiro, PUC-Rio. **Phd**.
- Singh, M. P. and M. N. Huhns (2005). Service-oriented Computing: Semantics, Processes, Agents, Wiley.

- Staa, A. v. (2006). Engenharia de Software Fidedigno. Monografias em Ciência da Computação, n. 13/06. PUC-Rio. Rio de Janeiro, Pontifícia Universidade Católica do Rio de Janeiro.
- Stelling, P., C. DeMatteis, et al. (1999). "A fault detection service for wide area distributed computations." Cluster Computing 2(2): 117-128.
- Sterling, L. S. and E. Y. Shapiro (1994). The Art of Prolog: Advanced Programming Techniques, MIT Press.
- Team, C. P. (2006). CMMI for Development, Version 1.2, Carnegie Mellon University.
- Tel, G. (2000). Introduction to Distributed Algorithms, Cambridge University Press.
- Thomas, D., D. H. Hansson, et al. (2006). Agile Web Development with Rails, Pragmatic Bookshelf.
- Van Solingen, R. and E. Berghout (1999). "The Goal/Question/Metric Method." A practical guide for quality improvement of software development.
- Vincenzi, A. M. R., A. S. Simão, et al. (2006). "Muta-Pro: Towards the Definition of a Mutation Testing Process." Journal of the Brazilian Computer Society 12.
- Weyns, D., A. Omicini, et al. (2007). "Environment as a first class abstraction in multiagent systems." Autonomous Agents and Multi-Agent Systems 14(1): 5-30.
- Wikipedia. (2007). "Tenerife disaster." Retrieved June, 11th, 2007, 2007.
- Xu, J., B. Randell, et al. (1995). Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery.
- Yi, X. and K. J. Kochut (2004). "Process composition of web services with complex conversation protocols: a colored petri nets based approach." Proc. of the Design, Analysis, and Simulation of Distributed Systems Symposium at Adavanced Simulation Technology Conf: 141–148.