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). Nesta seção 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.

Esta seção 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).

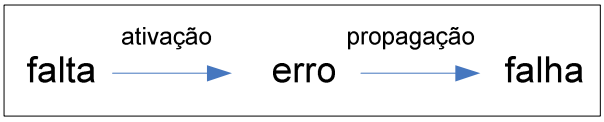

Figura 21 – Relacionamento entre falta, erro e falha (Avizienis, Laprie et al. 2004)

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 conseqüência, 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 conseqüência 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 nesta seção 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 desta seção o foco da discussão 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.

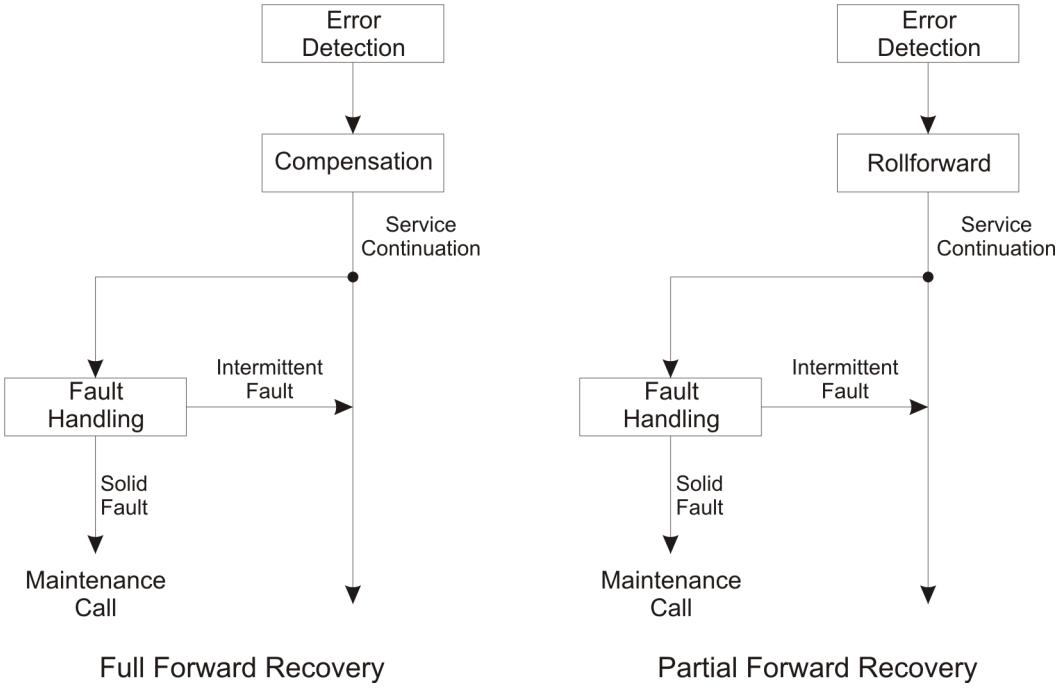

Figura 22 – Estratégia *Forward Recovery (Avizienis, Laprie et al. 2004)*

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.

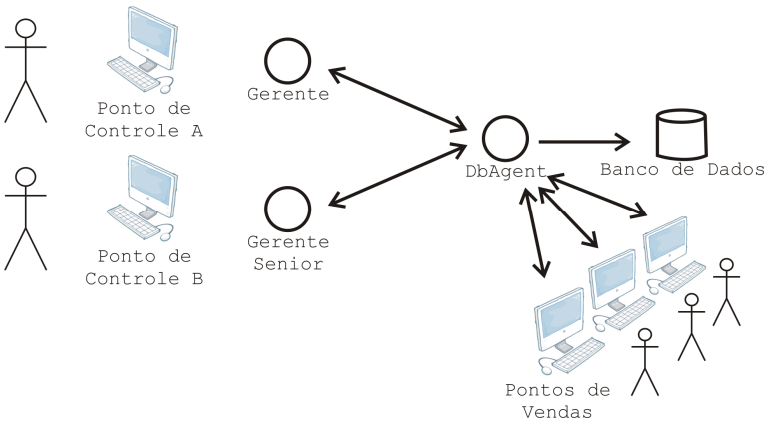

Figura 23 – Sistema de Controle de Vendas

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.

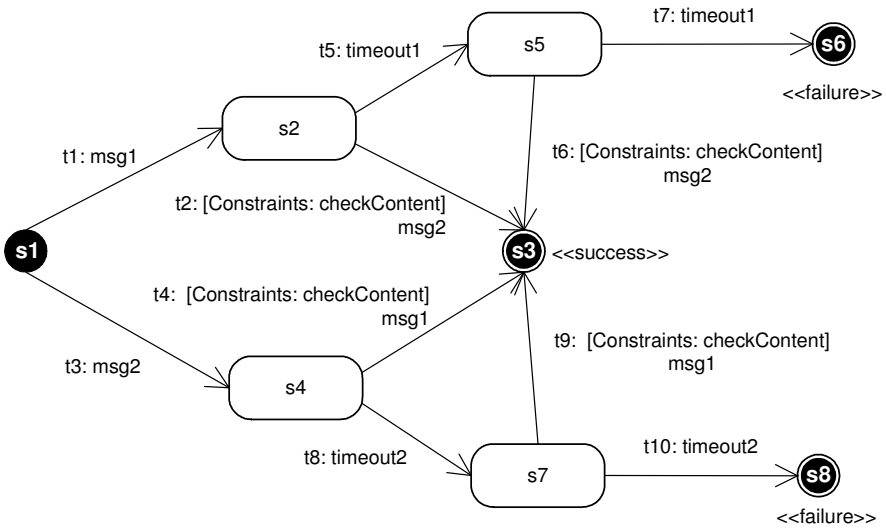

Figura 24 – Protocolo de Interação do Sistema de Controle de Vendas

```
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 efetive 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 conseqüência 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 **detecção 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 da seção 6.1 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.

Nesta seção, 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.

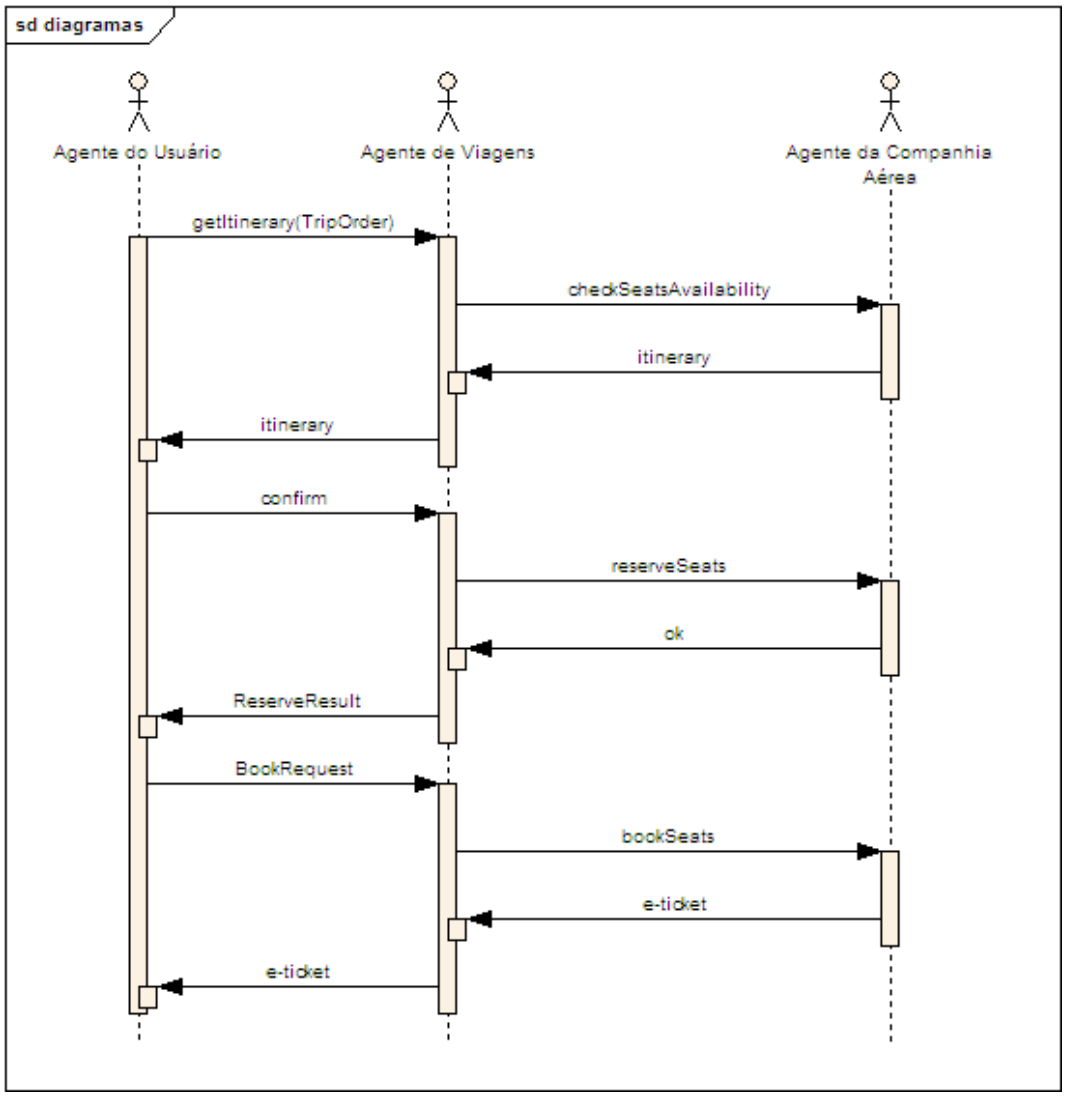

Figura 25 – Diagrama de Seqüência do Estudo de Caso

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.

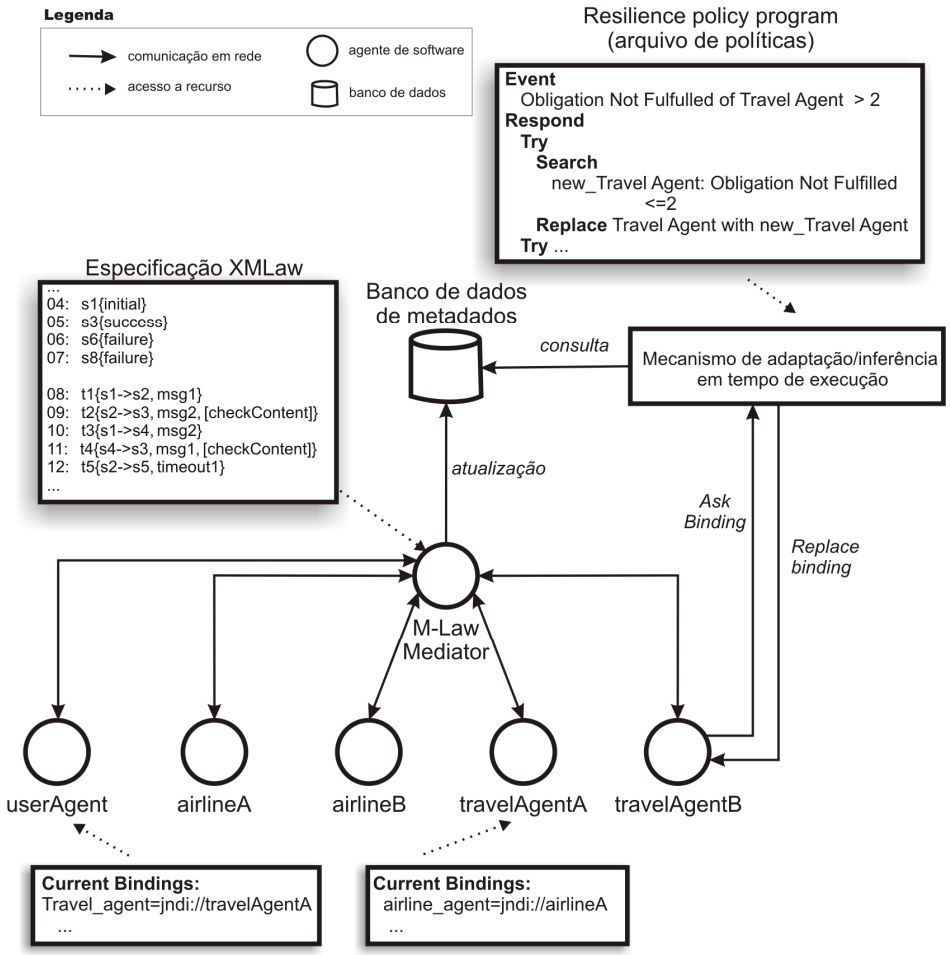

Figura 26 – Arquitetura do Estudo de Caso de DepEx

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 XMLaw que ele lê. No contexto da aquisição de dados, a especificação XMLaw 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={"Toronto", "New York", "London", "Tokyo", "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 cincos 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={"Toronto", "New York", "London", "Tokyo", "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.

O agente da companhia aérea envia a mensagem *itinerary-1* para o agente de viagens. Isto significa que a companhia aérea está dizendo "existe um assento disponível". Então, ativa-se um *clock* para contra o tempo. Além disso, também ativa-se a obrigação *hold-seat* para o agente da companhia aérea.
WHEN (t3, transition_activation) ACTIVATE hold-seat-clock, hold-seat
Se o prazo que a companhia aérea possui para aguardar o assento expirar (evento *clock_tick*), então a obrigação *hold-seat* não precisa mais ser cumprida, ou seja, o agente da companhia aérea pode responder com uma mensagem *not-available.*
WHEN (hold-seat-clock, clock_tick) DEACTIVATE hold-seat
Se a companhia aérea responde com um *result-ok* a uma requisição *reserveSeats*, 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 *t15* só dispara se a obrigação *hold-seat* estiver desativada. Se a agência de viagens envia uma mensagem *not-available* enquanto a obrigação ainda estiver ativa, então o evento *norm_not_fulfilled* 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 *action updateHoldSeatMetadata* é responsável pelas informações do contexto tais como a identificação do agente e da obrigação (neste caso *hold-seat*) 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 da 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.

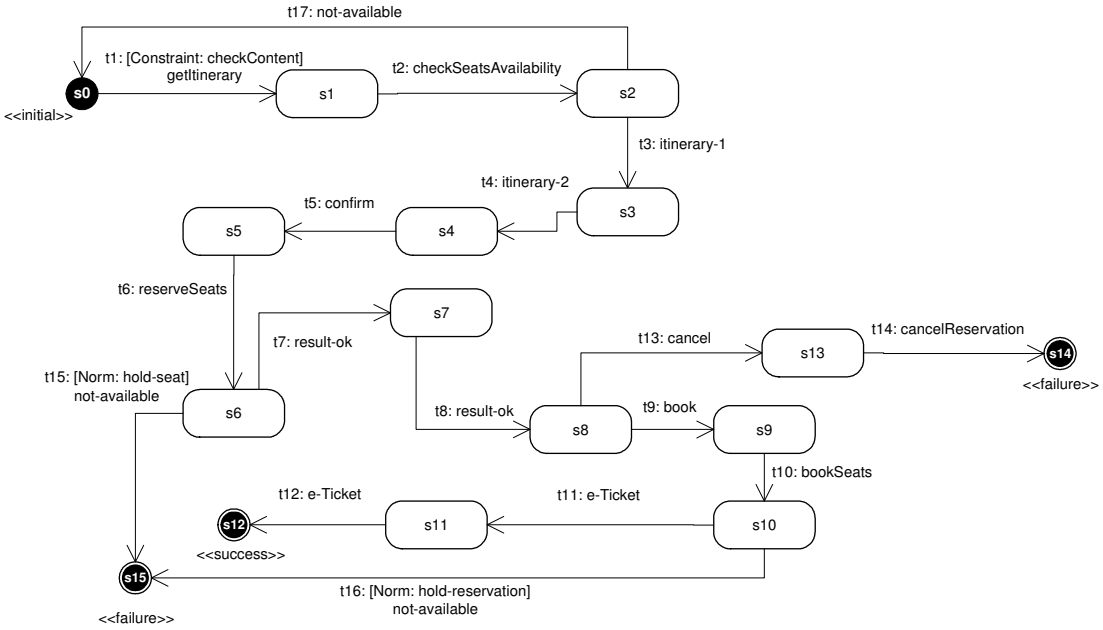

Figura 27 – Protocolo de Interação do Estudo de Caso de DepEx


```
// 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.

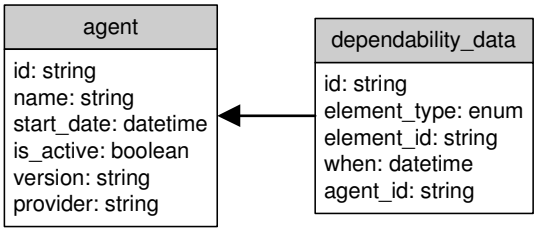

Figura 28 – Modelo Entidade-Relacionamento do Banco de Dados de Metadados

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 valo true se o agente está em execução	when – data e hora do resgistro 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.

Resultset 1

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

Resultset 1

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 nesta seção, 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 desta seção:

(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

Na seção 6.2 , 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 capitulo 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.