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.

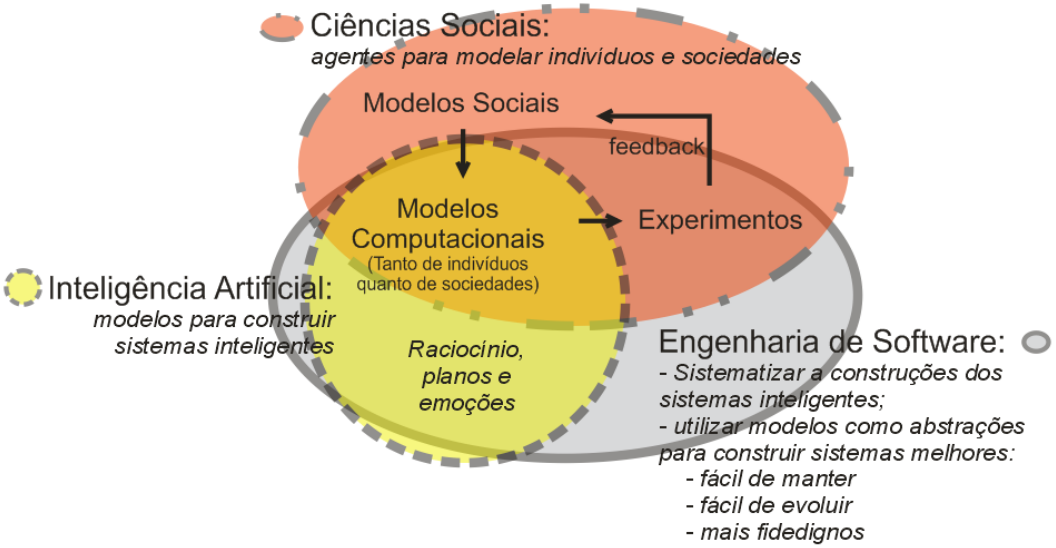

Figura 1 – Governança sob a Ótica das Disciplinas de Inteligência Artificial, Ciências Sociais e Engenharia de Software

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 OMG[4] 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

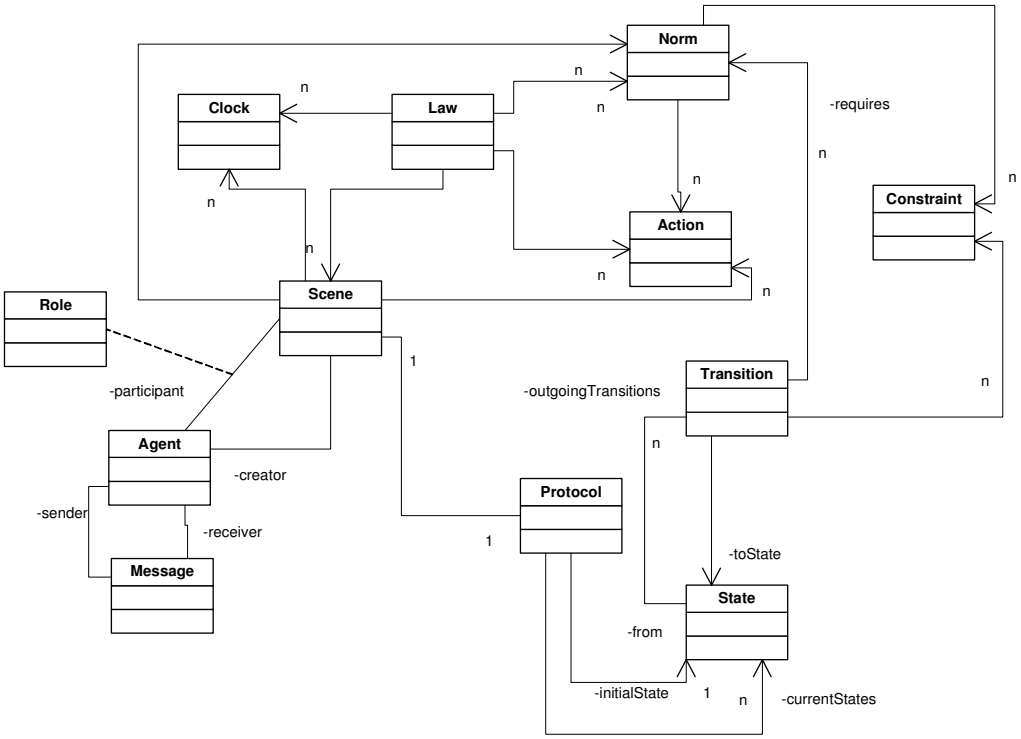

Figura 2 – Modelo Conceitual do XMLaw

	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 especificas 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: compreende 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 ciclicamente 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 indepedentemente 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 especifico 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 explicita 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_elapsed*.

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.

sucessful_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:

$$M= (E, S, T, s0, F)$$

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:

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

s0: o estado s0, definido como estado inicial tal que s0 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 quando ao tipo de evento que pode ativar um elemento. Esta informação é fornecida na especificação da lei.

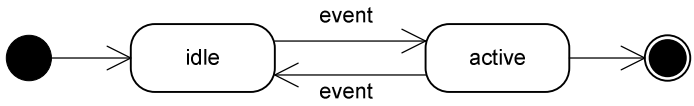

Figura 3 – Ciclo de Vida de um Elemento de Lei

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
sucessful_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

Contraint = 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.