

Isaac Simões Araújo Pereira

Drools Situation: uma Abordagem
Baseada em Regras para Detecção de
Situações

Vitória - ES

30 de Março de 2012

Isaac Simões Araújo Pereira

Drools Situation: uma Abordagem Baseada em Regras para Detecção de Situações

Monografia apresentada da para
obtenção do grau de Bacharel em Ciência da
Computação pela Universidade Federal do
Espírito Santo.

Orientadora:

Prof^a. Dr^a. Patrícia Dockhorn Costa

UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA

Vitória - ES
30 de Março de 2012

Monografia de Projeto Final de Graduação sob o título “Drools Situation: uma Abordagem Baseada em Regras para Detecção de Situações”, defendida por Isaac Simões Araújo Pereira em 30 de Março de 2012, em Vitória, Estado do Espírito Santo, pela banca examinadora constituída por:

Prof^a. Dr^a. Patrícia Dockhorn Costa
Orientadora

Prof. Dr. João Paulo Andrade Almeida
Examinador

Izon Thomas Mielke
Examinador

DEDICATÓRIA

Dedico este trabalho aos meus pais e familiares.

AGRADECIMENTOS

Primeiramente, agradeço aos meus pais, Maria das Graças e Johnny George, pelo apoio irrestrito e dedicação, fazendo dos meus objetivos, os seus, durante a minha formação acadêmica.

Aos amigos que conheci durante o curso, que não são meros colegas. Seu apoio e encorajamento foram fundamentais durante o desenvolvimento deste trabalho.

À minha orientadora, Patrícia Dockhorn Costa, que em vários momentos acreditou mais em mim do que eu mesmo. Sem ela, este momento não seria possível, bem como a minha continuidade acadêmica também não seria.

RESUMO

Este trabalho objetivou desenvolver um modelo de execução de detecção de situações e realizá-lo na plataforma *JBoss Drools* na forma de um módulo de gerência de situações para suporte ao desenvolvimento de aplicações sensíveis ao contexto. No processo, um estudo sobre plataformas de regras foi realizado, com atenção particular sobre o *JBoss Drools*, cujo resultado guiou o design da ferramenta *Drools Situation*, de acordo com os principais requisitos desejados à identificação de situações.

Palavras-chave: Contexto, Sensibilidade a Contexto, Detecção de Situações, *JBoss Drools*, *Drools Situation*

ABSTRACT

This work aimed to develop an execution model to detect situations and realize it in the *JBoss Drools* platform as a situation manager module in order to support the development of context-aware applications. In this process, a study about rule platforms was conducted, with particular attention on the *JBools Drools* platform, which the results guided the design of the *Drools Situation* tool, according to the key elements required to situation identification.

SUMÁRIO

1	INTRODUÇÃO	10
1.1	MOTIVAÇÃO.....	10
1.2	OBJETIVO.....	10
1.3	ABORDAGEM	11
1.4	ESTRUTURA	11
2	SENSIBILIDADE AO CONTEXTO	13
2.1	CONCEITOS GERAIS.....	13
2.1.1	Contexto	14
2.1.2	Aplicação Sensível ao Contexto	14
2.1.3	Fontes Contextuais.....	16
2.1.4	Situações.....	17
2.1.5	Modelagem Contextual.....	17
2.2	CAPTURA, COMPOSIÇÃO E GERÊNCIA DE CONTEXTO.....	18
2.2.1	Técnicas de Identificação de Situações	19
2.3	TRABALHOS RELACIONADOS.....	21
2.3.1	Uma Linguagem e Modelo de Execução para Detecção de Situações Ativas.....	21
2.3.2	Inferência de Situações para Usuários Móveis: Uma Abordagem baseada em Regras.....	24
2.4	CONSIDERAÇÕES	27
3	REGRAS, EVENTOS E DROOLS	29
3.1	SISTEMAS BASEADOS EM REGRAS	29
3.1.1	Base de Conhecimento	30
3.1.2	Base de Regras.....	30
3.1.3	Máquina de Inferência	31
3.2	CASAMENTO DE PADRÕES (<i>PATTERN MATCHING</i>).....	32
3.2.1	O Algoritmo RETE	33
3.3	O CICLO DE INFERÊNCIA.....	33
3.3.1	Resolução de Conflitos.....	34
3.4	JBOSS DROOLS	36

3.4.1	Expert: Conhecimento em Drools.....	37
3.4.2	<i>Fusion</i> : Processamento de Eventos Complexos	41
3.5	CONSIDERAÇÕES	46
4	MODELO DE DETECÇÃO DE SITUAÇÕES	47
4.1	INSPIRAÇÕES DA MODELAGEM DE SITUAÇÕES CONTEXTUAIS	47
4.2	MODELO DE MÁQUINA DE SITUAÇÕES.....	49
4.2.1	Conceituação Estrutural	50
4.2.2	Modelo Comportamental de Situações.....	51
4.2.3	Relações Temporais de Situação.....	56
5	DROOLS SITUATION.....	57
5.1	CONCEPÇÃO E ARQUITETURA GLOBAL.....	57
5.1.1	<i>JBoss Drools</i> Como Plataforma: Motivação	57
5.1.2	Arquitetura e Uso.....	58
5.1.3	Especificação de Situações.....	59
5.1.4	Multiplicidade de Papéis Situacionais.....	60
5.1.5	Raciocínio Temporal com Situações	61
5.2	ARQUITETURA INTERNA E IMPLEMENTAÇÃO.....	63
5.2.1	Sobrescrição de Operadores Temporais.....	66
5.3	PADRÕES DE SITUAÇÕES EM <i>DROOLS SITUATION</i>	66
5.3.1	Situação Intrínseca.....	67
5.3.2	Situação Relacional.....	68
5.3.3	Situação de Relação Formal	69
5.3.4	Situação de Situações.....	70
5.3.5	Combinação de Situações.....	71
5.4	CONSIDERAÇÕES	72
6	CONCLUSÃO E TRABALHOS FUTUROS.....	74
6.1	TRABALHOS FUTUROS.....	74
	REFERÊNCIAS	76
	ANEXO A – EXPRESSÕES DOS OPERADORES TEMPORAIS DE EVENTOS	78
	ANEXO B – CÓDIGO DO <i>SITUATIONHELPER</i>.....	81

1 INTRODUÇÃO

1.1 MOTIVAÇÃO

O visível crescimento da computação móvel têm posto em evidência áreas de pesquisa tais como a computação pervasiva e a computação sensível ao contexto. Esta última envolve a utilização de informações sobre o ambiente do usuário para oferecer algum tipo de reatividade ou adaptação às necessidades do mesmo.

A composição desse ambiente depende, basicamente, de informações cedidas proativamente pelo usuário ou de sensores capazes de tornar um determinado aspecto observável para a aplicação. A sua reatividade se dá a partir da identificação de estados de interesse sobre o contexto, porém, a identificação desses estados torna-se complexa à medida que a granularidade dos dados contextuais aumenta em relação ao cenário a se identificar, e não mais a caracterizam semanticamente.

Trabalhos como (COSTA, ALMEIDA, et al., 2006), (COSTA, 2007) e (COSTA, ALMEIDA, et al., 2007), avançam no sentido de propor suporte conceitual e arquitetural para a detecção de situações, explorando o paradigma de regras, fazendo uso de plataformas como *Jess*, e padrões de processamento de eventos, como por exemplo, o padrão *Event-Condition-Action (ECA)*. Em adição a estes trabalhos, uma nova plataforma de regras, *JBoss Drools*, tem mostrado potencial como plataforma de realização para os estudos citados, principalmente por integrar, numa só ferramenta, suporte à regras e eventos, em alguns aspectos aderentes aos conceitos propostos.

1.2 OBJETIVO

Este trabalho tem por objetivo estabelecer uma visão sobre situações e explorar a plataforma *JBoss Drools* a fim de desenvolver uma ferramenta, sobre esta

última, que oferece suporte natural ao modelo de situações proposto, tonando-o um *first class citizen* da mesma. Tal visão deve priorizar a facilidade na especificação de situações, integridade e a expressividade da linguagem quanto às correlações sobre suas propriedades, principalmente as temporais. Além disso, sua realização deve manter aderência com a visão de situações proposta em (COSTA, 2007), sendo capaz de reproduzir os padrões nele discutidos. Para isso é necessário: (i) compreender o paradigma da sensibilidade ao contexto; (ii) explorar a arquitetura e funcionamento de sistemas baseados em regras, e do *Drools* em particular; (iii) conceber cenários por meio da ferramenta desenvolvida, a fim de avaliar sua eficiência e empregabilidade.

1.3 ABORDAGEM

A abordagem utilizada neste trabalho consiste em i) estudar fundamentação teórica acerca da sensibilidade ao contexto, regras, eventos e plataformas que as empreguem, ii) levantar e analisar os requisitos para ferramentas de detecção de situações, iii) projetar e implementar um módulo de detecção de situações que atenda os requisitos levantados iv) empregar o módulo desenvolvido para implementar de cenários baseados em padrões de situações de (COSTA, 2007) a fim de identificar problemas de expressividade ou alguma outra possível deficiência do projeto.

1.4 ESTRUTURA

Os Capítulos 2 e 3 apresentam a fundamentação teórica deste trabalho. No Capítulo 2 são discutidos os principais conceitos referentes à área de sensibilidade ao contexto, identificação de situações e trabalhos relacionados. Em 3 são apresentados conceitos gerais sobre sistemas baseados em regras e um aprofundamento na plataforma *JBoss Drools*.

O Capítulo 4 apresenta um modelo de execução para detecção e gerência de situações orientadas a regras e eventos. Discute a influência dos modelos

contextuais de (COSTA, 2007) e formas de especificar regras para identificação de situações.

O Capítulo 5 propõe uma implementação do modelo de detecção de situações, discutido no Capítulo 4, sobre a plataforma *Drools*, concebendo o *Drools Situation*, e demonstra sua utilização na realização de alguns cenários situacionais propostos em (COSTA, 2007).

Por fim, no Capítulo 6 são apresentadas as conclusões e indicações de trabalhos futuros.

2 SENSIBILIDADE AO CONTEXTO

A proposta da sensibilidade ao contexto é utilizar informações sobre o ambiente do usuário, seu contexto, a fim de oferecer algum tipo de comportamento reativo ou pró-ativo às aplicações. Este paradigma traz consigo um novo nível de interação: contexto-aplicação, que auxilia a produção de serviços com maior aderência às necessidades do usuário.

Um dos recursos exploráveis da sensibilidade ao contexto é caracterização complexa do usuário, isto é, a expansão das informações sobre seu ambiente baseado em caracterizações mais primitivas, possibilitando a identificação de, e racionalização sobre, aspectos mais abstratos de interesse do seu ambiente. Porém, a identificação destes estados, ou situações, não é uma tarefa simples, constituindo um dos desafios das aplicações sensíveis ao contexto, e pelo mesmo motivo, alvo, na literatura, de um grande número abordagens de solução.

Este capítulo apresenta conceitos relacionados à área de sensibilidade ao contexto e técnicas de identificação de situações. Ele está organizado da seguinte forma: a Seção 2.1 apresenta definições relacionadas à área de sensibilidade ao contexto, utilizadas neste trabalho; a Seção 2.2 discute os principais requisitos associados a composição de contexto e inferência de situações. Por fim, a Seção 2.3 discute alguns trabalhos relacionados à detecção de situações na área de computação pervasiva.

2.1 CONCEITOS GERAIS

As seções seguintes introduzem os conceitos relativos ao tema de sensibilidade ao contexto.

2.1.1 Contexto

De acordo com (DEY, 1999) “contexto é qualquer informação que pode ser usada para caracterizar a situação das entidades (uma pessoa, um lugar, ou um objeto) que são relevantes para uma aplicação, incluindo o próprio usuário e a própria aplicação”. Portanto, de acordo com essa definição, se um pedaço de informação é utilizado para caracterizar um participante da interação usuário-aplicação, esta informação é contexto. Exemplos comuns de contexto são a localização de um usuário, a frequência cardíaca de um paciente, a orientação de um dispositivo móvel, etc.

2.1.2 Aplicação Sensível ao Contexto

De acordo com (COSTA, 2007), uma aplicação sensível ao contexto pode ser definida como “uma aplicação distribuída, cujo comportamento é afetado pelo contexto do usuário”. Em (DEY, 2001), a definição é semelhante, porém, também descreve o que é oferecido por uma aplicação sensível ao contexto: “um sistema é sensível ao contexto se ele utiliza contexto para prover informação e/ou serviços relevantes para o usuário, em que a relevância depende das tarefas dos usuários”.

Portanto, pode-se entender Aplicação Sensível ao Contexto como uma aplicação que, para oferecer informações ou serviços para o usuário, faz uso de contexto. Uma aplicação pode, por exemplo, utilizar a localização do usuário a fim de informá-lo sobre a disponibilidade de serviços próximos, como restaurantes, lojas, etc. A Figura 2.1 mostra uma visão intuitiva de um usuário (*user*) e seu contexto (*user's context*), e uma aplicação sensível ao contexto (*context-aware application*) (focando em um usuário apenas).

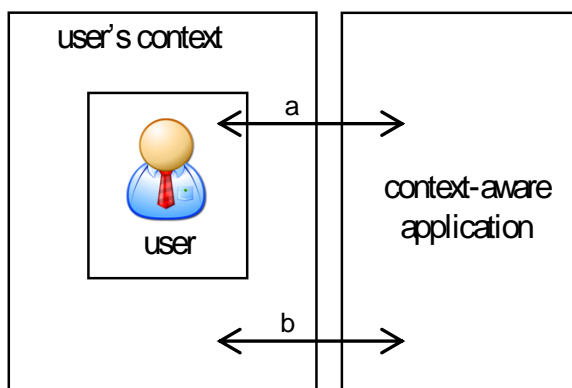


Figura 2.1 – Visão de aplicação sensível ao contexto interagindo com o usuário e seu contexto (COSTA, 2007).

Na Figura 2.1, a seta “a” mostra que o usuário e a aplicação sensível ao contexto interagem. Semelhantemente, a seta “b” mostra que o contexto do usuário e a aplicação sensível ao contexto também interagem. Uma interação representada pela seta “a” pode ser exemplificada por entradas fornecidas pelo usuário à aplicação, ou pelo usuário utilizando serviços fornecidos pela aplicação. As interações do tipo “b” possibilitam a aplicação sensível ao contexto capturar condições contextuais do usuário.

Em (DEY, 2001), as aplicações sensíveis ao contexto são categorizadas segundo a finalidade a que elas se propõem:

- Apresentação de informação e serviços para o usuário;
- Execução automática de um serviço para o usuário;
- Adição de contexto à informação para futuras recuperações.

Como exemplo da primeira categoria, tem-se (WEIßENBERG et al., 2008), que propõe uma abordagem para um sistema baseado na web voltado para oferta de serviços inteligentes, baseados em contexto, a uma larga quantidade de usuários. O sistema, denominado FLAME2008, teria como foco grandes eventos sociais, como os Jogos Olímpicos de 2008, ou mesmo, a Eurocopa de 2008, oferecendo serviços aos seus usuários de acordo com suas demandas. Na segunda categoria estão aplicações que realizam ações com base em informações contextuais. Nesta categoria, pode-se citar *Drools Emergency Service Application*, proposto em

(SALATINO, 2011), no qual que o acionamento de alarmes e solicitação de serviços de ambulância são gerados a partir da detecção de situações de risco. As condições de risco são avaliadas com base no monitoramento remoto de pacientes, sendo adquiridas algumas condições contextuais do usuário, como sinais cardíacos, e localização. A terceira categoria pode ser exemplificada por uma aplicação que analisa e classifica transações bancárias em busca de situações de fraude. O contexto de uma sessão de *internet banking*, por exemplo, tempo de acesso, qual a localização do dispositivo que efetuou o acesso, quais transações foram realizadas e quais contas estiveram envolvidas, é armazenado e pode ser agregado a outras informações para atender futuras consultas ou análises.

2.1.3 Fontes Contextuais

As informações contextuais são disponibilizadas para as aplicações sensíveis ao contexto por meio das chamadas *fontes de contexto*, que fornecem informações de interesse para aplicação, permitindo que estas operem independentemente de como as informações são capturadas.

Geralmente, *fontes de contexto* são manifestadas por sensores, que proativamente monitoram dados do ambiente, por exemplo, localização, temperatura, etc. Ainda assim, o próprio usuário pode ser reconhecido como *fonte de contexto*, ao prover manualmente informações contextuais, embora, do ponto de vista da formação de contexto não há distinção entre contextos obtidos manualmente ou via sensores.

Uma fonte de contexto deve ser capaz de aplicar abstrações sobre dados brutos capturados, principalmente no que se refere a sua granularidade. Por exemplo, uma aplicação de deseja determinar a localidade de um usuário, pode obter coordenadas brutas de um sensor GPS e traduzi-las na informação de que o usuário se encontra em uma cidade. Contextos “brutos”, geralmente detectados diretamente via sensores, são denominados *contextos primitivos*, enquanto que os contextos processados, de mais alto nível, são denominados de *contextos compostos*.

2.1.4 Situações

(COSTA, 2007) define situação como “um estado particular de interesse para aplicações cujos seus constituintes podem ser uma combinação de entidades e de suas condições contextuais”. Dessa forma, situações são abstrações de estados de contexto, ou grupo de contextos, sobre certas condições, e em geral, seu valor semântico assume papel mais relevante que a sua própria caracterização, seus constituintes ou condições impostas estes, que a instanciaram em primeiro lugar. O que significa dizer que, o simples estabelecimento de uma situação já é, por si só, uma informação determinante para afetar o comportamento de uma aplicação.

A construção de aplicações sensíveis ao contexto se beneficia de situações contextuais, pois, como sistema distribuído, seus nós podem compartilhar artefatos de contexto de alto nível, reduzindo a carga geral de processamento e troca de dados, inclusive, limitando-a a determinados nós produtores ou consumidores de situação em algumas arquiteturas. Por exemplo, suponha uma aplicação sensível ao contexto composta de dois módulos: um que realiza diagnóstico (situação) de pacientes, e outro, que administra um tratamento baseado em um diagnóstico. O segundo módulo, está alheio às condições que determinam certo diagnóstico, atendo-se apenas a como tratar a ocorrência do mesmo.

Situações em nível de instância possuem propriedades temporais, dado que a própria instanciação de uma situação representa a sua ocorrência no domínio em determinado instante e durante determinado tempo. Tais propriedades denotam a possibilidade de se estabelecer relações temporais entre situações. São exemplos de questões de interesse das aplicações, que envolvem o fator tempo, ao manipular situações saber: A que instante uma situação se estabeleceu? Por quanto tempo ele se estendeu? Ou se uma situação do tipo A ocorreu antes de alguma do tipo B.

2.1.5 Modelagem Contextual

Em (COSTA, 2007) é definido um conjunto de abstrações de modelos de contexto baseado em teorias de modelagem conceitual. De acordo com (GUIZZARDI, 2005), modelos conceituais são representações abstratas de um

determinado domínio, independentemente de um *design* específico ou escolhas tecnológicas. Segundo (COSTA, 2007), “a modelagem conceitual de contexto deve anteceder o *design* detalhado das aplicações sensíveis ao contexto, da mesma forma que as análises devem anteceder o *design* detalhado de um sistema de informação”. Um modelo conceitual de contexto abstrai de como o contexto é detectado, providenciado, aprendido, produzido e/ou usado.

Durante o processo de fundamentação dos artefatos de modelagem contextual (COSTA, 2007) identificou padrões recorrentes na modelagem conceitual de cenários contextuais, e posteriormente situacionais. Esses padrões identificam propriedades importante como composição de contextos complexos, e até mesmo temporalidade entre situações, como abordado na seção anterior.

Embora particularidades da modelagem contextual proposta por (COSTA, 2007) não tomem parte, de forma direta, do escopo deste trabalho, é de interesse que os artefatos produzidos por este permitam a plena reprodução dos padrões situacionais presentes em (COSTA, 2007). A seção 5.3 deste trabalho reflete esta preocupação.

2.2 CAPTURA, COMPOSIÇÃO E GERÊNCIA DE CONTEXTO

O desenvolvimento de uma aplicação sensível ao contexto é uma tarefa desafiadora, pois, geralmente, este tipo de aplicação deve ser capaz de: (i) detectar o contexto do ambiente, (ii) observar, coletar e compor as informações contextuais utilizando diversos meios (frequentemente sensores), (iii) automaticamente detectar mudanças relevantes no contexto, (iv) reagir a estas mudanças, adaptando seu comportamento ou invocando (composição de) serviços. Estes desafios exigem que a plataforma, na qual será desenvolvida a aplicações sensíveis ao contexto, forneça mecanismos capazes de atender aos requisitos relacionados a este tipo de aplicação.

Assim, o fator chave para o desenvolvimento de aplicações sensíveis ao contexto é fazer uma ponte sólida entre a obtenção dados brutos de sensores, ou nível mais baixo de contexto, e a identificação do que, de fato, se deseja estar ciente

sobre as aplicações, as suas *situações*. Entre estas duas pontas têm-se a necessidade de meios de descrição de relações entre contextos, para composição de contextos complexos, situações, e principalmente de formas de gerenciá-las, isto é, maneiras de interpretar tais descrições e identificá-las sobre todo o domínio, representar suas ocorrências e perceber quando não mais ocorrem e, além disso, de oferecer estes artefatos de forma abstrata para os serviços interessados.

As questões apresentadas anteriormente são englobadas pela área de *Detecção de Situações*. Em (YE et al, 2011) são destacados tópicos de pesquisa acerca deste tema, tais como, de que maneira pode-se definir primitivas lógicas para se especificar situações, operações que manipulem suas propriedades, e como estas especificações podem ser assimiladas por especialistas ou aprendidas através de dados para treinamento, envolvendo aprendizado de máquina. Além delas há também o questionamento de como inferir situações a partir de dados contextuais brutos mantendo a consistência e integridade do conhecimento adquirido. Na seção seguinte são apresentadas algumas abordagens possíveis a estes problemas.

2.2.1 Técnicas de Identificação de Situações

Entre os fatores que dificultam a identificação de situações estão: imprecisão, incompletude, expressividade na descrição de relações e manipulação de situações. O estudo realizado em (YE et al, 2011) discute técnicas comumente aplicadas na detecção de situações na área da computação pervasiva, à luz dos fatores citados previamente. Neste estudo, as abordagens são classificadas entre dois supertipos: as *técnicas baseadas em especificação* e as *técnicas baseadas em aprendizado*.

As *técnicas baseadas em especificação* (Figura 2.2) tipicamente partem da construção de modelos de situações com um conhecimento especialista *a priori* para, então, racionalizar sobre entradas de dados provenientes de sensores, interpretando-as. (YE et al, 2011) destaca o uso de *Lógicas Formais*, *Lógicas Espaço-temporais*, *Teoria de Situações* e *Ontologias*, como exemplos deste tipo de abordagem. Estendendo estas soluções, são citadas: a aplicação de *Lógica Fuzzy* e da *Teoria de Evidência de Dempster-Shafer*, como formas de lidar com

determinados níveis de imprecisão, incompletude e inconsistência dos dados capturados via sensores.

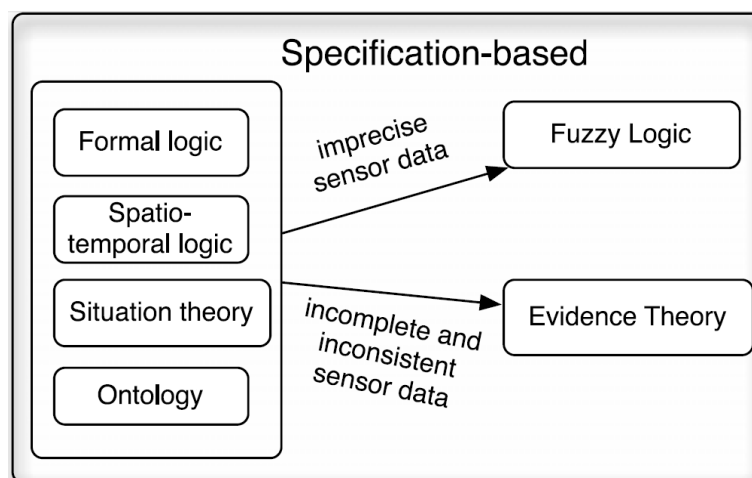


Figura 2.2 – Técnicas Baseadas em Especificação (YE, et al. 2011)

Como contraponto, temos as *técnicas baseadas em aprendizado*, motivadas por questões relacionadas à escalabilidade, complexidade e inconsistência (em alto grau) de sensores, que, como afirma (YE et al, 2011), “tornam as técnicas baseadas em especificação, dependentes de um *a priori expert knowledge*, impraticáveis”. Destacam-se, neste escopo, a aplicação de Redes Bayesianas, Hidden Markov Models (HMMs), ou Modelos Ocultos de Markov, Redes Neurais. Tais técnicas aplicam aprendizado de máquina para alcançar um refinamento no processo de identificação, porém, exigem uma grande quantidade de treinamento e parametrização.

Baseando-se no legado científico que motivou este trabalho, podendo-se citar: (COSTA, ALMEIDA, 2006), (COSTA, 2007) e (COSTA, ALMEIDA, 2007), as abordagens *baseadas em especificação* são, naturalmente, o foco de discurso aqui, especialmente a *Lógica Formal*, *Lógica Espaço-Temporal* e *Ontologias*. De forma geral, elementos destas técnicas permearão a abordagem deste trabalho como (i) especificação lógica de situações por meio de regras (*Lógica Formal*); (ii) raciocínio temporal sobre situações (*Lógica Espaço-Temporal*), de acordo com (ALLEN, 1981); (iii) modelagem contextual (*Ontologias*), como vista em 2.1.5.

2.3 TRABALHOS RELACIONADOS

A seguir são apresentados alguns trabalhos que abrangem a área de sensibilidade ao contexto com ênfase em técnicas para de detecção de situações contextuais.

2.3.1 Uma Linguagem e Modelo de Execução para Detecção de Situações Ativas

(ADI, 2002) aponta como grande deficiência de sistemas reativos e proativos a dificuldade em assimilarem a distância entre os eventos passíveis de identificação e os casos que exigem a reação do mesmo. (ADI, 2002) define tais casos como *situações*, equivalendo-as a tipos de padrões complexos que se apresentam sobre um fluxo de eventos, ressaltando como motivação de seu trabalho, a incapacidade das ferramentas de apoio a aplicações orientadas a eventos em lidarem com eventos complexos, sendo de pouca utilidade para situações.

(ADI, 2002) apresenta um modelo de execução para suporte a situações cujos conceitos são aplicados sobre um cenário de controle de tráfego. Neste cenário, o tráfego é controlado através de sinais de trânsito presentes entre cruzamentos e faixas de pedestres, como os da Figura 2.3. A unidade de controle recebe diversos tipos eventos que vão desde a informação de que um pedestre, ou veículo aguarda a mudança de sinal até a notificação de sobrecarga de trânsito no local.

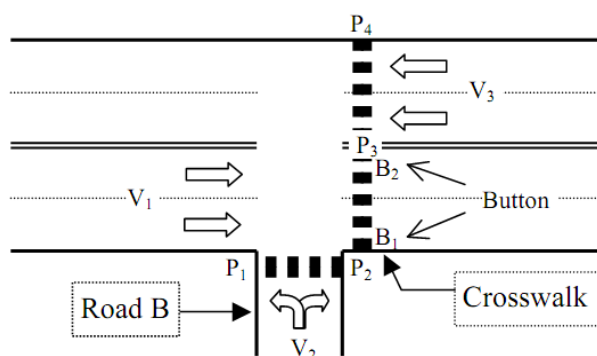


Figura 2.3 – Controle de Tráfego (ADI, 2002)

Abaixo são definidos os principais conceitos do modelo de (ADI, 2002):

Modelo de Evento

Um evento é uma ocorrência atômica causada por uma transição entre estados do domínio. Ele é materializada através de uma *instância de evento*. Por sua vez, uma *classe de evento* descreve propriedades comuns a um conjunto de instâncias de eventos num nível abstrato. Além de classes de eventos, o modelo permite a definição de classes de *agrupamento de eventos*, que coletam instâncias que possuam algum predicado comum, pode-se citar, no cenário em questão, eventos provenientes de veículos que estão na mesma pista ou eventos originados de pedestres que aguardam no mesmo sinal, por exemplo.

Lifespan

(ADI, 2002) conceitua *lifespan* como um determinado intervalo temporal no qual a detecção de uma situação é relevante, e é delimitado por dois eventos, *initiator* e *terminator*, respectivamente. A definição de um *lifespan*, ou classe de *lifespan*, descreve o conjunto de eventos capazes de disparar uma instância de *lifespan* ou encerrá-la, bem como o tempo máximo para qual ela é válida. Um *lifespan* pode ser encarado como um predicado básico, primário, para que um tipo de situação ocorra. (ADI, 2002) não clarifica as motivações por trás da definição de *lifespans*, porém, fica subentendido que a sua existência propicia uma melhor distribuição de processamento ao longo do tempo de execução, o que também justifica outras decisões de design, como o conceito de agrupamento de eventos, discutido anteriormente.

Por exemplo, supondo que uma situação de interesse da unidade controle de tráfego seja o estado em que o sinal de um cruzamento se manteve vermelho por 5 minutos (*lightIsRedFiveMinutes*), ela é relevante e detectada somente quando existir um *lifespan* iniciado pelo evento no qual o sinal se torna vermelho e encerrado quando este volta a ser verde.

Contexto

O Contexto de (ADI, 2002) é uma noção semântica distinta da definição de contexto em aplicações sensíveis ao contexto, abordadas neste capítulo. Ambas descrevem conhecimento parcial sobre estado do domínio, no entanto, enquanto a última é centrada no estado de entidades particulares do mesmo, a primeira se mantém com foco no todo, além disso, a sua aplicabilidade é como escopo de detecção de situações, estendendo a função que a definição de *lifespan* já denotava. Assim, um contexto é composto de um *lifespan*, um agrupamento de eventos e um predicado.

Reutilizando o exemplo de situação do tópico anterior, temos que um contexto de detecção da situação *lightsRedFiveMinutes* seria composto de um *lifespan* delimitado pelo intervalo em que o sinal se manteve vermelho, e juntam-se a ele: o agrupamento de eventos cujas instâncias provém do cruzamento do sinal em questão, e um predicado que verifica se a unidade de controle não foi sobrescrita por algum operador. É importante ressaltar que um mesmo contexto de detecção pode ser compartilhado entre mais de um tipo de situação.

Situação

A definição de uma situação consiste de uma expressão algébrica sobre uma combinação de contextos e *coleção de eventos*. Uma situação é composta por uma *expressão de situação* e uma *expressão de desencadeamento*, ou disparo. A primeira determina as condições para a detecção de situações e os eventos que a causaram. A última define as ações a serem tomadas como resultado de uma detecção. (ADI, 2002) não aborda desta forma, porém, as duas expressões formam uma regra, ou classe de regras a qual poderíamos tratar como *regras de situação*.

O processo final de detecção se baseia na análise de *coleções de eventos*. Uma coleção de eventos retém eventos candidatos a dispararem certas situações, e, por sua vez, são particionados em listas candidatas, que reúnem eventos que desempenham um mesmo *papel* na detecção de um tipo de situação, e. g., numa situação de congestionamento de uma via, eventos que

reportam sobre veículos desempenham um papel, enquanto eventos que reportam sobre pedestres, outro. O controle de uma lista candidata define condições de entrada, de remoção e de reuso de instâncias para novas detecções.

A linguagem para composição de *expressões de situação* é formada por uma combinação de um operador e qualificadores, um predicado. Os operadores suportados são: operadores de junção (*conjunction*, *disjunction*, *sequence*, *strictSequence*, *simultaneous*, e *aggregation*), operadores de seleção (*first*, *until*, *since*, e *range*), operadores de afirmação (*never*, *sometimes*, *last*, *min*, *max*, e *unless*), e operadores temporais (*at*, *after*, e *every*). Por exemplo, considerando uma situação que representada por um pedestre que pressiona três vezes consecutivas o botão (*buttonPressed*) para travessia de uma faixa (*buttonPressedThreeTimes*), pode-se utilizar o operador *sequence* para selecionar eventos cuja ocorrência seja sequencial, neste caso, três ocorrências do evento *buttonPressed*. A expressão completa é apresentada em 2-1.

1	<situationExpression>
2	<Sequence>
3	<event name="buttonPressed" qualifier="last"/>
4	<event name="buttonPressed" qualifier="last"/>
5	<event name="buttonPressed" qualifier="last"/>
6	</Sequence>
7	</situationExpression>

2-1 - Expressão para Situação *buttonPressedThreeTimes*

2.3.2 Inferência de Situações para Usuários Móveis: Uma Abordagem baseada em Regras

No trabalho apresentando em (GOIX, et al., 2007) é destacado o aumento da aplicação de dispositivos móveis como plataformas multissensor, e como esse fator facilita a contextualização do usuário. A partir desta premissa, (GOIX, et al., 2007) propõe uma arquitetura de coleta de dados contextuais, inferência baseada em regras e distribuição de situações sobre usuários de aplicações móveis.

A proposta consiste em utilizar combinações de dados, ora provenientes de dispositivos do usuário, ora provenientes de centrais de dados, e através destes, inferir situações a serem fornecidas para serviços interessados. A Figura 2.4 apresenta um esquema desta arquitetura.

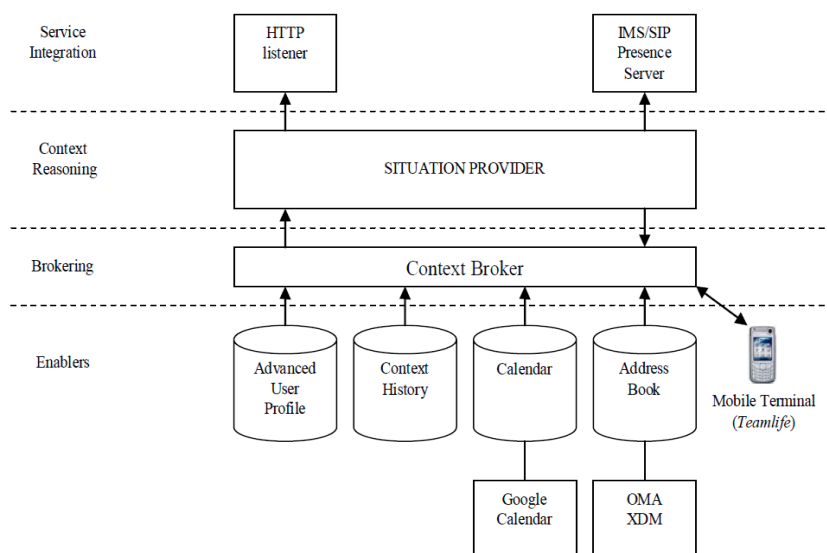


Figura 2.4 – Arquitetura Global do Processo de Inferência de Situações (GOIX, et al. 2007)

Os componentes desta arquitetura estão distribuídos em quatro camadas:

- Camada de Viabilizadores (*Enablers*): refere-se aos provedores de contexto, com destaque para o *teamlife*, a aplicação móvel que fornece, periodicamente, dados como dispositivos *bluetooth* próximos, localização geográfica e identificação do dispositivo do usuário. A aplicação suporta várias configurações, podendo, por exemplo, limitar o envio de dados apenas quando algum o número de dispositivos *bluetooth* próximos mudar.
- Camada de Intermediação (*Brokering*): Nela o *Context Broker* agrega estas diversas fontes promovendo uma padronização das informações contextuais, provendo-as num formato homogêneo e processável.
- Camada de Inferência de Contexto (*Context Reasoning*): É capaz de inferir situações de interesse baseadas nos artefatos contextuais concretizados pelo *Context Broker*. A Figura 2.5 detalha a arquitetura do *Situation Provider*.

- Camada de Integração de Serviços (*Service Integration*): Apresenta uma interface para subscrição de notificações sobre situações de um conjunto de usuários, por parte de serviços web.

Os diversos provedores de contexto abastecem a camada adjacente com dados contextuais brutos, muitas vezes heterogêneos, de difícil associação entre si. O *Context Broker*, por sua vez, agrega estas diversas fontes promovendo uma padronização das informações contextuais, provendo-as num formato homogêneo e processável. O *Situation Provider* compreende um *Sistema Baseado em Regras* (tipo de sistema abordado com detalhes em 3), cujos fatos são dados contextuais providos pelo *ContextBroker* e cuja base de regras é composta de por regras de situação, que seguem a forma retratada em 2-2.

1	(defrule [context pre-conditions] =>
2	(assert ([new situation]))
3	(retract ([no longer valid situation]))

2-2 - Template de Regra de Situação (GOIX, et al., 2007)

Estas regras podem então ser compostas de maneira incremental, tendo como pré-condições dados contextuais como localização ou status social e inferindo a partir destes a atividade atual de um usuário, e, tipicamente, retratando (invalidando) sua situação anterior.

(GOIX, et al., 2007) utiliza *ContextML* para representar os dados contextuais. A integração com o *Situation Provider* se dá através da transformação *ContextML-RuleML*. *RuleML* é a linguagem utilizada pelo motor de regras que compõe o *Situation Provider*, ele é capaz de processar fatos em *RuleML* produzindo, então, novos fatos, ou invalidando antigos. No caso, essa produção também se dá em *RuleML*, e a transformação inversa é necessária, para que o *Context Broker* agregue os novos dados contextuais inferidos.

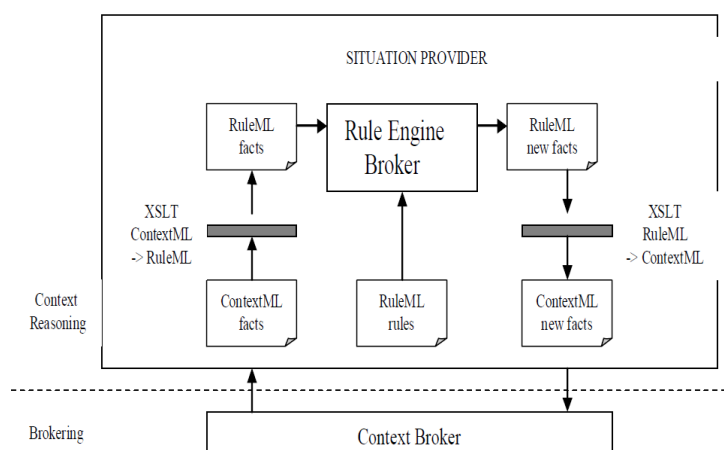


Figura 2.5 – Situation Provider: Processo de Inferência de Situações (GOIX, et al., 2007)

O ciclo de inferência da é definido a seguir:

1. Fatos, em *ContextML*, são extraídos do *Context Broker*;
2. A Base de Conhecimento do *Situation Provider* é abastecida com os fatos extraídos, já em formato *RuleML*, após a aplicação de transformação via *XSL*;
3. A Máquina de Regras é executada e há, possivelmente, a inferência de novos fatos;
4. Os novos fatos produzidos são notificados ao *Situation Provider*, transformados em *ContextML*, via *XSL* inversa, e adicionados ao *ContextBroker*.

2.4 CONSIDERAÇÕES

A realização de situações sob o paradigma baseado em regras, de acordo com o proposto em (COSTA, 2007), constitui a inspiração inicial para o desenvolvimento deste trabalho. Vale ressaltar ainda, a influência dos trabalhos relacionados citados na seção anterior, (ADI, 2002) e (GOIX, et al. 2007), cuja influência se estende principalmente sobre a definição de um modelo particular de execução de situações, como será visto em 4, e sobre sua implementação, apresentada em 5, respectivamente.

Neste capítulo foram apresentados os principais fundamentos teóricos que suportam este trabalho, no entanto, no decorrer do mesmo, outros fundamentos importantes, como o paradigma de regras e eventos complexos, surgiram eventualmente em algumas discussões, porém, sem apresentações formais. O capítulo seguinte abordará devidamente estes conceitos, a fim de complementar esta fundamentação.

3 REGRAS, EVENTOS E DROOLS

Sistemas baseados em regras são orientados por um paradigma que se utiliza de conjuntos de precondições e ações, também chamadas de regras, para obter certo comportamento especialista, ou de ação seletiva. Sendo um tema crucial no contexto deste trabalho. Neste capítulo são apresentadas características gerais acerca dessa classe de sistemas, posteriormente, atendo-se com detalhes a plataforma *JBoss Drools (JBoss Rules)*, base de realização deste trabalho e exemplo de plataforma que emprega o paradigma em foco.

Este capítulo está estruturado da seguinte forma: a seção 3.1 apresenta os componentes gerais de um sistema baseado em regras, na seção 3.2 são abordados algoritmos de *rule matching* usados por máquinas de regras., em 3.3 é apresenta o ciclo de inferência de regras, em 3.4 é apresentado o *JBoss Drools*, suas particularidades e principais módulos e, finalmente, em 3.5 são feitas considerações sobre os conceitos apresentados à luz dos requisitos para identificação de situações.

3.1 SISTEMAS BASEADOS EM REGRAS

De modo geral, uma regra é uma diretriz que aponta um curso usual, costumeiro, ou generalizado de ação ou comportamento sob certas condições. Programaticamente falando, ao construirmos uma aplicação, estamos invariavelmente aplicando regras ao caracterizarmos seu comportamento, decidindo uma sequência determinística de ações caso determinadas condições sejam satisfeitas.

Em abordagens ditas tradicionais, i. e., abordagem estruturada e afins, esta caracterização se dá em meio à estrutura da aplicação, denotadas por declarações *if-e/se* da própria linguagem de programação. Tem-se então, arquitetura e comportamento fatalmente compartilhando as mesmas estruturas de código.

Sistemas Baseados em Regras (SBR) propõem uma separação clara entre estrutura da aplicação e suas regras de negócio, com foco no último aspecto, de maneira que a manutenibilidade das regras é uma vantagem imediata desta abordagem.

Tipicamente, um SBR é composto por três elementos:

- Base de Conhecimento;
- Base de Regras;
- Máquina de Inferência.

Essa subdivisão é recorrente nos sistemas baseados em regras atuais, como *iLog*, *CLIPS*, *Jess* e *Drools*, entre outros. Uma das vantagens dessa arquitetura é oferecer uma clara separação entre os dados (conhecimento sobre o domínio) e o controle (como o conhecimento é aplicado).

3.1.1 Base de Conhecimento

A base de conhecimento (*working memory*) consiste de asserções sobre um determinado domínio de discurso. Essas asserções são também conhecidas como fatos. Por exemplo, supondo uma aplicação médica baseada no paradigma em questão, uma asserção desse domínio poderia ser a informação de que *um paciente em particular apresenta certo sintoma*, em 3-1 temos uma representação deste fato em lógica de predicados.

1	$R_1: paciente(x) \wedge sintoma(y) apresenta(x, y)$
---	--

3-1 - Representação de Fato Lógica de Predicados

3.1.2 Base de Regras

A base de regras contém todas as definições de regras aplicáveis, representando, dessa forma, todo o comportamento da aplicação. Ela define de que forma o conhecimento armazenado na *working memory* deverá ser usado, e todas essas expectativas estão concentradas no seu elemento básico: a regra.

Uma regra é formada por uma tupla de precondição e consequência (se-então). A primeira, também conhecida como *Left Hand Side* (abreviadamente LHS), define as premissas impostas aos fatos para que a mesma seja satisfeita. A última, a *Right Hand Side* (RHS), é o conjunto de ações a serem executadas caso as precondições sejam atendidas.

Uma declaração de regra geralmente segue a forma:

1	Regra X
2	Se <cond_1>, <cond_2>, ... , <cond_N>
3	Então <acao_1>, <acao_2>, ... , <acao_M>

3-2 - Descrição de Regra

Onde os elementos condicionais (**cond_N**) na LHS podem apresentar relações disjuntivas (ou) e conjuntivas (e) entre si. Em muitos casos os SBR oferecem linguagens para especificação de regras, onde essas condições, também conhecidas como padrões (*patterns*), podem ser descritas em termos inspirados na lógica de predicados.

Analogamente ao que foi exemplificado sobre a base de conhecimento, um exemplo de regra desse mesmo domínio: ***se** algum paciente apresenta determinado sintoma, **então** deve-se aplicar certo tratamento sobre o mesmo*. Note que as ações de uma regra podem intervir na base de conhecimento, incluindo, alterando ou retratando fatos. Por exemplo, ao desencadear a ação da regra citada acima, isto é, a aplicação de um tratamento médico, é bem possível, e esperado, que se obtenha uma melhora do paciente, e que assim, o mesmo não apresente mais o sintoma, invalidando a assertiva.

3.1.3 Máquina de Inferência

A máquina de inferência é o coração de um sistema baseado em regras. Ela é responsável por buscar, analisar e gerar novo conhecimento num processo de encadeamento lógico que permite extrair conclusões ou confirmar suposições de acordo com os fatos e regras existentes.

Quanto ao método de inferência, há duas abordagens predominantes: o encadeamento regressivo e o encadeamento progressivo. O encadeamento

regressivo (*backward chaining*) consiste no lançamento de uma hipótese, e daí, na busca por fatos que a sustentem, tornando a suposição, enfim, numa assertiva. Nesse sentido, diz-se que é uma técnica orientada a objetivos (hipótese). *Por sua vez*, o encadeamento progressivo (*forward chaining*) é um método que se baseia em fatos conhecidos a fim de estabelecer novas conclusões. Por tal motivo, diz-se que é uma abordagem *orientada a dados*.

Suponha que se queira determinar o estado de saúde (**doente** ou **saudável**), de uma pessoa (a_1), sob a base de regras abaixo, dado que a mesma apresenta 39 graus de temperatura.

R_1 : se *temperatura_acima*($x, 37$) então *febre*(x)

R_2 : se *febre*(x) então *sintomático*(x)

R_3 : se *sintomático*(x) então *doente*(x)

R_4 : se não(*sintomático*(x)) então *saudável*(x)

Aplicando-se *forward chaining* no exemplo anterior, parte-se do fato de a temperatura de a_1 estar acima de 37 graus. Este ativa a regra R_1 , adicionando um novo fato à base (*febre*(a_1)), que por sua vez ativa R_2 , inferindo: *sintomático*(a_1), que, por fim, desencadeia a ativação de R_3 e obtém-se a informação desejada: *doente*(a_1).

Neste momento, passamos a considerar o *forward chaining* como a estratégia padrão para as máquinas de inferência supostas ou apresentadas posteriormente neste capítulo, isto se deve ao fato de uma melhor adequabilidade entre esta abordagem e regras de produção, que desencadeiam ações, em contraponto ao *backward chaining*, voltado para regras de derivação, cuja aplicação se dá no estabelecimento de conclusões lógicas.

3.2 CASAMENTO DE PADRÕES (*PATTERN MATCHING*)

A abordagem utilizada no casamento de padrões entre regras e fatos é o grande diferencial de uma máquina de regras. A eficiência em detectar a ativação de uma regra e dispará-la é determinante, pois, neste ponto, uma abordagem

convencional, de **ifs** e **elses** aninhados num código procedural, já o faz imediatamente.

3.2.1 O Algoritmo RETE

O Rete (*ree-tee*) é um algoritmo de *pattern matching* proposto por Charles Forgy em (FORGY, 1974), e amplamente aplicado em sistemas de produção, cujo objetivo é evitar a abordagem de busca exaustiva, linear e pouco eficiente, tanto da *working memory* quanto da base de regras.

Consiste na construção de uma rede de nós que reflete a base de regras como um todo. Cada nó da rede representa um *pattern* presente em uma ou mais regras, e guarda fatos que durante sua execução satisfaçam tal *pattern*. A rede *RETE* forma uma estrutura de árvore. O RETE sacrifica a memória para priorizar a velocidade, e seu desempenho é, teoricamente, independente da quantidade de regras do sistema. Resoluções parciais são armazenadas em nós intermediários da Árvore RETE, poupando o sistema de ter de reavaliar todos os fatos a cada alteração da *working memory*, dessa forma o RETE permite que o processamento se concentre no Delta entre o estado pré-alteração e pós-alteração da *working memory*.

Cada elemento da *working memory* entra na RETE através do seu nó raiz (*Root Node*), e são propagados através de nós descendentes, à medida que satisfazem os respectivos *patterns*, e se encerram num nó terminal (*Terminal Node*). Este caminho (*Path*) representa a LHS de uma regra em si. Os fatos que concluem o *path* formam o contexto de ativação da regra, definido em detalhes na seção seguinte.

3.3 O CICLO DE INFERÊNCIA

A máquina de inferência de regras atua em ciclos que se iniciam na tentativa de **rule matching** e se encerram quando uma única regra é escolhida e disparada. Esse ciclo conhecido como *Ciclo de Reconhecimento-Ação* (Figura 3.1), se sucede até que o **rule matching** não acione mais nenhuma regra.

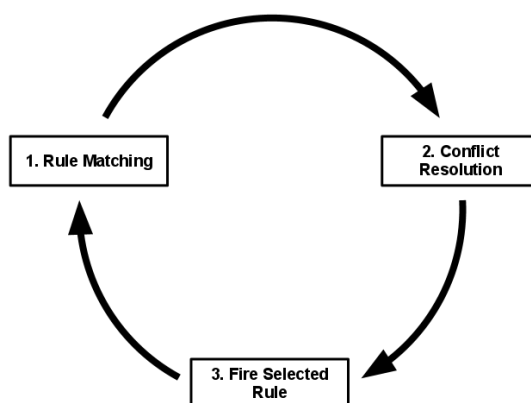


Figura 3.1 - Ciclo de Reconhecimento-Ação

Quando um **rule match** ocorre, isto é, quando os antecedentes de uma regra são atendidos, diz-se que houve uma instanciação de regra. É a evidência de que a regra está apta a ser disparada, ela carrega consigo a identificação da regra, e ainda mais importante, o contexto ao qual deve ser aplicada. Em alguns sistemas baseados em regras, a instanciação de uma regra é chamada de **ativação**, brevemente citada no escopo do algoritmo RETE.

Duas instâncias de regra são ditas idênticas quando:

- a) Possuírem em comum a mesma regra;
- b) Os antecedentes da regra foram satisfeitos pelo mesmo conjunto de fatos de ambas as instâncias no mesmo ciclo.

Assim, ao final de uma iteração de *rule matching* obtém-se um conjunto de ativações. Neste estágio, a máquina precisa definir qual ativação será, de fato, disparada. Por esse motivo o conjunto obtido é também conhecido como **conflict set**, ou conjunto de conflito, em tradução livre.

3.3.1 Resolução de Conflitos

A maioria dos sistemas baseados em regras apresenta ao menos um estratégia para lidar com o *conflict set*. Em (SASIKUMAR, 2007) são apresentadas três técnicas clássicas, são elas: especificidade, recência e refratariedade. Outra estratégia digna de menção é a saliência, presente em SBR comerciais como *Jess* e *Drools*. Abordaremos cada uma dessas estratégias em seguida.

Especificidade (Specificity)

Baseia-se na complexidade dos antecedentes da regra. Regras com maior número de premissas combinadas, isto é, as mais específicas, são privilegiadas.

Por exemplo, considere as duas regras:

R_1 : se febre(x) então anti_inflamatorio(x)

R_2 : se febre(x) e dengue(x) então não anti_inflamatorio(x)

E considere também os seguintes fatos:

F_1 : febre(a_1)

F_2 : dengue(a_1)

Aplicando as regras R_1 e R_2 sobre estes fatos, obtém-se ativação para ambas, no entanto, aplicando-se a especificidade, R_2 seria selecionada para disparo, pois suas precondições são um superconjunto das precondições de R_1 . Este exemplo é útil para demonstrar a efetividade do método da especificidade ao tratar exceções na base de regras.

Recência (Recency)

Trata-se de uma estratégia exclusiva para conflitos entre ativações de uma mesma regra. Com ela priorizam-se ativações cujos fatos são mais recentes. A ideia nesse caso é que regras disparadas com base em dados mais recentes são mais precisas e apresentam maior potencial para resolução efetiva do problema, portanto, são mais relevantes.

Refratariedade (Refractness)

A refratariedade é determinante para evitar a produção de conhecimento redundante, contribuindo, evidentemente, para a eficiência da máquina de regras. Ela impede que regras se apliquem consecutivamente. Caso uma instanciação da regra tenha sido ativada anteriormente, novas ativações, idênticas àquela, não serão desconsideradas nos ciclos posteriores.

Saliência (Salience)

Propriedade inerente à regra, que permite ao autor da mesma declarar sua prioridade em relação às outras regras, geralmente a prioridade é diretamente proporcional ao seu valor. Dentre as abordagens apresentadas é a única que permite a ação direta do usuário sobre a resolução, embora se deva mencionar que alguns SBR, o *Jess* sendo um deles, ofereçam a possibilidade de implementação de estratégias particulares.

3.4 JBOSS DROOLS

(BALI, 2009) apresenta O *JBoss Drools* como uma plataforma de integração de lógica de negócio, baseada na linguagem Java, que une regras, processamento de eventos complexos (CEP) e gerência de processos em uma só ferramenta. É um projeto open-source suportado pelo *JBoss* e a *Red Hat Inc.* sobre a licença Apache 2.0. Nesta seção foram consideradas as características e propriedades da versão 5.2 do mesmo.

Desde o seu nascimento, em 2001, como uma máquina de regras que realizava *pattern matching* por busca linear, o *Drools* agregou novas habilidades na forma de módulos. Hoje são cinco ao todo: *Expert*, *Fusion*, *Flow*, *Planner* e *Guvnor*. O *Drools Expert* consiste numa máquina de regras que materializa os conceitos apresentados previamente neste capítulo, sendo assim, um SBR clássico, que utiliza uma versão orientada a objetos do algoritmo *RETE*, e constitui a base de toda a plataforma. O módulo *Fusion* agrega ao *Expert* o potencial do processamento de eventos complexos. O *Flow* oferece a capacidade de especificar processos de negócio (workflow) e integrá-los a regras. O *Planner* é voltado à resolução de problemas de agendamento e alocação de recursos, tipicamente NP-completos. Por fim, o *Drools Guvnor* consiste em um gerenciador de repositórios de bases de conhecimento, modelos, regras e processos de negócio, que fornece controle e ferramentas para edição de todo este conhecimento armazenado.

Um elemento essencial é seu *plugin* para o *Eclipse IDE* que auxilia no desenvolvimento de seus projetos, fornecendo uma interface para especificação de

conhecimento (regras e fatos), com checagem sintática e possibilidade *debugging* de regras.

3.4.1 Expert: Conhecimento em Drools

A máquina de regras *Drools* segue a mesma arquitetura de sistemas baseados em regras, descrita anteriormente. A base de conhecimento, a base de regras e a máquina de inferência estão presentes aqui e interagem da mesma forma, ressaltando que o conhecimento, ou fato, é representado por referências (chamados de *FactHandles*) a objetos *Java*, dessa forma, todo atributo primitivo, ou relacionamento entre objetos, estabelecido em nível de instância, representa uma porção válida de conhecimento para a máquina.

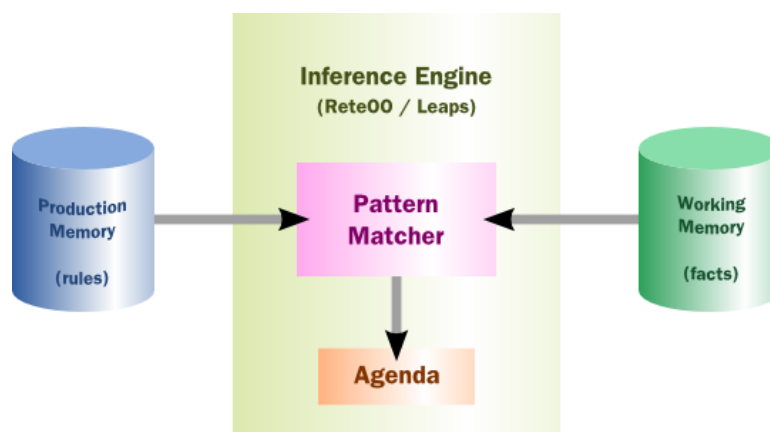


Figura 3.2 - Arquitetura do *Drools Expert* (DROOLS EXPERT, 2011)

O ciclo de reconhecimento-ação ocorre no contexto de sessões de execução independentes, instâncias de uma *StatefulKnowledgeSession*. Cada sessão possui a sua própria base de regras, e gerencia o conjunto de fatos, *FactHandles*, que compõem a sua memória de trabalho.

Especificação de Regras

O *Drools* introduz o seu próprio meio de especificação de regras, a DRL (*Drools Rule Language*). As regras são descritas em arquivos, no formato **.drl*, e consistem no principal recurso para a construção de uma sessão de conhecimento.

Uma regra em DRL é composta de blocos LHS e RHS, e, além disso, um bloco para meta-atributos, sendo o mais destacável a *saliência* (*saliency*), ou prioridade da

regra. A LHS é descrita através de um dialeto que permite especificar padrões e restrições sobre objetos de classes Java (estritamente *POJOs*) e seus atributos, e relacioná-los logicamente (*and*, *or*, *not*, *exists*, etc), contando, inclusive, com operadores de conjuntos (*memberof*, *contains*, etc) e funções de agregação (*accumulate*) em sua composição. A RHS é especificada em código Java.

Um trecho de um arquivo DRL onde se define uma regra:

1	rule "Paciente com Febre"
2	when
3	Paciente(temperatura > 37)
4	then
5	System.out.println("Paciente apresenta febre");
6	end

3-3 - Exemplo de regra em Drools

Em 3-3 temos a declaração da regra "Paciente com Febre". Seu bloco **when** (LHS) apresenta uma estrutura sintática chamada de **pattern** (Linha 3). Um **pattern** aplica uma seleção de tipo sobre os fatos e, internamente, é composto das chamadas **constraints**, que consistem de restrições lógicas aplicadas sobre os indivíduos do tipo selecionado pelo **pattern**.

Um **LHS pattern** consiste num agrupamento de restrições, que, por sua vez, são expressões booleanas, aplicáveis sobre instâncias da classe ao qual o **pattern** representa. O **pattern** em 3-3 apresenta uma única restrição: que a *temperatura* (atributo da classe *Paciente*) seja maior que o valor inteiro 37. No bloco **then** (RHS) invoca-se uma ação de *output* do sistema para a mensagem "Paciente apresenta febre".

Na LHS permite-se declarar identificadores para referenciar objetos capturados numa ativação, tornando-os visíveis e manipuláveis no escopo da RHS, são as chamadas **binding variables**. Em 3-4 temos uma versão da regra anterior que faz uso de um identificador, **\$nome**, dentro de um **pattern**.

1	rule "Paciente com Febre"
2	when
3	Paciente(\$nome: nome, temperatura > 37)
4	then
5	System.out.println("Paciente" + \$nome + " apresenta febre");
6	end

3-4 - Regra em Drools com identificador declarado na LHS

Operadores da *KnowledgeSession*

Regras de Produção eventualmente afetam a base de conhecimento, criando, alterando ou removendo fatos. *Drools* suporta um conjunto de operadores no escopo da RHS para essa finalidade. São eles: *insert*, *update* e *retract*, respectivamente.

Por exemplo, a regra declarada em 3-5 utiliza o operador *insert*. Ela tem por premissa todo o paciente cuja temperatura assume um valor acima de 37 graus e que não exista um registro de febre cujo *febril* (atributo da classe Febre) corresponda ao paciente em questão.

```

1 rule "Paciente com Febre"
2   when
3     $p: Paciente(temperatura > 37)
4     not (exists(Febre(febril==$p)))
5   then
6     insert(new Febre($p));
7   end

```

3-5 - Regra "Paciente com Febre"

Na RHS de 3-5, um objeto Febre é instanciado e, enfim, inserido na *working memory* através do *insert*, tornando-se um fato.

Como contraponto, tem-se a regra declarada em 3-6, a qual exemplifica o *retract*. Ela considera como premissa todo o caso de febre e um paciente, que coincida com o paciente *febril* deste caso em questão, cuja temperatura assume um valor igual ou abaixo de 37 graus.

```

1 rule "Paciente recuperado de Febre"
2   when
3     $f:Febre($p: febril)
4     Paciente($p == this, temperature <= 37)
5   then
6     retract($f);
7   end

```

3-6 – Regra "Paciente recuperado de Febre"

A RHS de 3-6 recolhe o fato Febre da *working memory*, dado que o paciente referenciado por *febril* não está mais neste estado, logo, um conhecimento não mais válido.

Por sua vez, o operador **update** é aplicado quando alterações em atributos de um fato foram motivadas por alguma regra. Sua forma de utilização na RHS é sintaticamente semelhante as dos outros operadores, retratados nos exemplos anteriores. Vale ressaltar, que estas operações, como interventoras da base de conhecimento, são passíveis de provocarem execuções do ciclo de reconhecimento-ação.

Além das operações de conhecimento típicas, o *Drools* oferece um recurso chamado de *inserção lógica*, por meio dele é possível garantir a integridade ao conhecimento. Consiste em associar fatos produzidos por meio de regras com a validade das premissas que o originaram. Significa dizer que, um fato inserido logicamente só existe enquanto as condições para a ativação que o originou se mantiverem verdadeiras, sendo recolhido pela própria *engine* quando não forem mais. No exemplo anterior, foram necessárias duas regras para inferir a condição de febre de um paciente, a primeira para detectá-la e a segunda para recolher o fato quando a condição de febre deixar de ser válida. Considerando-se o mesmo cenário, desta vez empregando inserção lógica (via operador *insertLogical*), tem-se:

1	rule "Paciente com Febre"
2	when
3	\$p: Paciente(temperatura > 37)
4	then
5	insertLogical (new Febre(\$p));
6	end

3-7 - Regra "Paciente com Febre" com Inserção Lógica

Dessa forma todo o cenário é sintetizado em uma só regra (3-7), muito semelhante à 3-5, embora mais simples, pois a verificação da prévia existência de um fato Febre para o paciente não é necessária. Este **pattern** era necessário em 3-5 para garantir que a regra não fosse disparada a cada alteração de temperatura (acima de 37 graus), inserindo múltiplos fatos de febre para o mesmo paciente, no entanto, no processo de inserção lógica a identidade dos objetos é verificada a fim de evitar inserções de fatos duplicados.

O mecanismo de inserção lógica será mais discutido nos próximos capítulos, dada a sua relevância para o desenvolvimento do módulo de detecção de situações.

3.4.2 *Fusion*: Processamento de Eventos Complexos

(ETZION, NIBLETT, 2010) definem evento como “uma ocorrência dentro de um domínio particular; algo que acontece, ou é notado como tendo acontecido neste domínio”. Sob uma descrição genérica, um evento pode ser conceituado como um *acontecimento notável*. Para propósitos computacionais, o termo evento é um objeto que representa ou registra este *acontecimento notável* no domínio e cujas propriedades inerentes a sua ocorrência são imutáveis. Por sua vez, um *evento complexo* consiste na abstração, através de agregação e composição, de um padrão de eventos, os quais são chamados de seus membros.

A área de Processamento de Eventos Complexos (CEP) lida com operações que incluem criar, transformar, abstrair eventos, e, além disso, fornecer meios para detecção, consumo e reação aos mesmos. A dimensão temporal tem papel de destaque neste processamento, dado que os atributos temporais inerentes aos eventos possuem forte valor semântico.

Para a plataforma em questão, tem-se, presumivelmente, uma abordagem CEP orientada a regras. Como visto nas seções anteriores, uma máquina de regras clássica é pautada por mudanças de estados e não por eventos, no entanto, no caso do *Drools*, ela foi estendida, tornando evento um artefato explícito de seu modelo. Dessa forma, ocorrências de eventos podem ser referenciadas como parte das condições de uma regra, o que, resumidamente, define o *Drools Fusion*.

Na perspectiva do *Drools Fusion*, eventos são apenas um tipo especial de fato, inclusive possuindo um *handler* específico (*EventFactHandle*), que apresenta características ou conceitos diferenciais como:

- (i) **Restrições temporais:** dado que regras envolvendo eventos tipicamente requerem alguma correlação temporal;
- (ii) **Imutabilidade:** Eventos são estáticos de modo que sua ocorrência e duração são conhecidas desde a sua instanciação, e nunca alteradas;
- (iii) **Ciclo de vida:** A *engine* é responsável por gerenciar a existência de eventos como fatos e removê-los quando não forem mais necessários.

Dentre os atributos de um *EventFactHandle* estão: o **instante de ocorrência** (*timestamp*), sua **duração** e **tempo de expiração**, este último consiste no período no qual o evento é relevante, fora dele o fato pode ser descartado pela máquina de regras. Vale ressaltar que, de forma geral, apenas a máquina lida diretamente com *FactHandles*, e que os atributos citados acima não são acessíveis, o que contribui para a garantia de sua imutabilidade.

O *Fusion* suporta dois tipos de eventos, no que se refere à temporalidade. De um lado, têm-se *eventos pontuais* representados pelo seu instante de ocorrência e com duração nula. De outro, há os *eventos baseados em intervalo* que ocorrem e se estendem num intervalo contínuo, portanto com duração não nula.

Especificação de Eventos

Por padrão, todo objeto inserido numa *KnowledgeSession* é tratado com um fato comum, de modo que, para que a *engine* lide com fatos como eventos é necessário declarar explicitamente quais classes assumirão este papel. Geralmente, a especificação de um tipo de evento é um simples mapeamento que atribui a uma classe Java pré-existente a semântica de eventos, no entanto, além dessa forma, é possível especificar classes inteiramente em *DRL*. Em 3-8 temos um trecho em *DRL* que declara a classe *LigacaoTelefonica*, cujo propósito é registrar eventos de chamadas telefônicas.

1	declare <i>LigacaoTelefonica</i>
2	//atributos
3	inicio: java.util.Date
4	duracao: long
5	numero: int
6	//meta-atributos
7	@role(event)
8	@timestamp(inicio)
9	@duration(duracao)
10	@expires(24h)
	end

3-8 - Declaração de evento

As linhas 3, 4 e 5 em 3-8 compõem a declaração da classe em si. O restante representa a especificação da mesma como um evento. Esta especificação é composta por quatro meta-atributos: *role*, *timestamp*, *duration* e *expires*. O meta-atributo *role* define qual tipo de *handler* será vinculado às instâncias do tipo

declarado. O *timestamp* define que valor a *engine* irá considerar como instante de ocorrência do evento e, caso não seja declarado, o instante de inserção do fato será considerado por padrão. A *duration* indica a duração do evento e quando não declarada indica um evento pontual (duração nula). O meta-atributo *expires* define um intervalo de validade dos eventos.

Relações Temporais entre Eventos

O *Fusion* implementa a semântica de eventos baseados em intervalo de tempo descrita por (ALLEN, 1981) na forma de treze operadores e seus respectivos complementos lógicos (negações). A Figura 3.3 retrata de forma simplista a aplicação destas operações entre as possíveis combinações de tipos de eventos. São eles: *before*, *after*, *meets*, *metby*, *overlaps*, *overlappedby*, *finishes*, *finishedby*, *starts*, *startedby* e *coincides*.

	Point - Point	Point - Interval	Interval - Interval
A Before B B After A	A ● B ●	A ●—● B ●	A ●—● B ●—●
A Meets B B Met by A		A ●—● ● B	A ●—● B ●—●
A Overlaps B B Overlapped by A			A ●—● B ●—●
A Finishes B B Finished by A		B ●—● A ●	B ●—● A ●—●
A Includes B B During A		A ●—● B ●	A ●—● B ●—●
A Starts B B Started by A		A ● B ●—●	A ●—● B ●—●
A Coincides B	A ● B ●		A ●—● B ●—●

Figura 3.3 - Relações temporais entre eventos suportadas pelo *Drools Fusion*

Suponha um cenário em que se deseja detectar chamadas telefônicas perdidas em caso do destino estar ocupado, cujas principais entidades são *LigacaoTelefonica*, definida em 3-8, e *Chamada*, um evento pontual que representa

a tentativa de ligação. Pode-se estabelecer como premissa que chamadas a um determinado número que ocorram **durante** (*during*) uma ligação já estabelecida a este mesmo número serão consideradas perdidas, pois o destino se encontra ocupado, dessa forma, tem-se 3-9.

1	rule “Chamada Perdida por Destino Ocupado”
2	when
3	\$ligacao: LigacaoTelefonica(\$num:numero)
4	\$chamada: Chamada(numero==\$num, this during \$ligacao)
5	then
6	insert (new ChamadaPerdida(\$chamada));
7	end

3-9 - Relações Temporais na DRL

Em 3-9 a relação temporal *during* é aplicada. De forma detalhada, este operador avalia a seguinte expressão:

1	\$ligacao.startTimeStamp < \$chamada.startTimeStamp &&
2	\$chamada.startTimeStamp < \$ligacao.startTimeStamp

3-10 - Operador *during*

No Anexo A – Expressões dos Operadores Temporais de Eventos estão descritas as expressões lógicas de todos os operadores temporais do *Drools Fusion*.

Streaming e Janelas de Eventos

O uso de *streams* de eventos, ou correntes de eventos é comum em plataformas CEP, eles são úteis para organizar eventos de origens distintas, classificá-los e ordená-los.

O Drools oferece suporte a *streams* de eventos através dos chamados *entry-points*, pontos de entrada de fatos na *working memory*. O uso de *entry-points* é facultativo, quando não declarados na *KnowledgeSession*, a *stream* de eventos consiste de uma fila padrão contendo todos os eventos já inseridos na *working memory*.

No exemplo da chamada telefônica, os eventos poderiam ser agrupados numa *stream* exclusiva para chamadas recebidas. A referência a fatos provenientes de *entry-points* é feita sobre um *pattern*, como em 3-11. Dessa forma, o escopo de

coleta do *pattern* se estende apenas sobre fatos do tipo *Chamada*, inseridos no entry-point “*ChamadasRecebidas*”.

1	Chamada() from entry-point “ChamadasRecebidas”
---	--

3-11 - Uso de *entry-point*

Um recurso decorrente do streaming de eventos é a definição de janelas de eventos. Janelas de eventos são amostragem de eventos delimitadas delimitadas pro fatores quantitativos ou temporais, isto é, pode ser obter conjuntos de eventos especificando seu tamanho, ou intervalo de coleta.. As janelas são úteis em casos como, por exemplo, calcular a média dos últimos saques bancários de um cliente, ou a frequência cardíaca durante o último.

Em 3-12 tem-se a regra que exemplifica o primeiro caso. A sintaxe para a declaração de janelas é feita sobre um *pattern* cujo tipo é um evento (linha 5), uma janela da forma: **window:length(10)** coleta os últimos 10 eventos que correspondem ao *pattern* associado à janela.

1	rule “Média de Saques”
2	when
3	\$cliente: Cliente()
4	\$mediaSaque: Number() from accumulate(
5	Saque(cliente==\$cliente, \$valor:valor) over window:length(10)
6	average(\$valor)
7)
8	then
9	\$cliente.setMediaSaque(\$mediaSaque)
10	update(\$cliente);
11	end

3-12 - Janela de Eventos por Tamanho

Em 3-12 têm-se a regra que exemplifica o segundo caso. Utilizando uma janela temporal da forma: **window:time(1m)**, que representa a coleta de todos os eventos que respeitam o *pattern* associado à janela, cuja ocorrência se deu no último minuto, e de forma geral, no intervalo [tempo atual - intervalo, tempo-atual].

1	rule “Frequência Cardíaca”
2	when
3	\$paciente: Paciente()
4	\$frequencia: Number() from accumulate(
5	\$batida: Batida(paciente==\$paciente) over window:time(1m)
6	count(\$batida)

7)
8	then
9	\$paciente.setFrequenciaCardiaca(\$frequencia)
10	update(\$paciente);
11	end

3-13 - Janela de Eventos por Tempo

3.5 CONSIDERAÇÕES

Podem-se destacar características de sistemas baseados em regras em geral que favorecem detecção de situações:

- Análise ativa sobre o domínio (*working memory*);
- Linguagem para especificação de premissas lógicas, padrões e restrições sobre objetos do domínio;
- Gerência lógica de fatos e manutenção da integridade do conhecimento pode favorecer o controle do ciclo de vida de situações.

Quanto ao caráter *CEP* do *Drools*, dado que situações e eventos compartilham características temporais, o modelo de eventos proposto pelo *Fusion* tangencia algumas propriedades desejáveis às situações, principalmente na parte de correlações de intervalos temporais, e sua aplicação para representação de situações é tentadora. Por outro lado, sua imutabilidade corrobora contra, uma vez que, situações são altamente dinâmicas, e sua manifestação ou não manifestação são um reflexo imediato do estado do domínio. No entanto, a composição de situações baseada em eventos delimitadores é uma possibilidade.

4 MODELO DE DETECÇÃO DE SITUAÇÕES

Este capítulo tem por objetivo apresentar uma visão intermediária, independente de plataforma, para a realização de situações, pautada sob os paradigmas de sistemas baseados em regras e processamento complexo de eventos e inspirada na fundamentação de situações contextuais proposta por (COSTA, 2007). No entanto, sua aplicação é independente de uma formalização contextual específica, à medida que oferece uma conceitualização leve ao nível de especificação estrutural, propositalmente não se comprometendo a nenhum paradigma de contexto no processo, mas, ao mesmo tempo preocupando-se para que a mesma possa expressar as situações segundo os padrões situacionais de (COSTA, 2007). Por fim, esta visão tem interesse na formalização de situações ao nível de instância, sua ocorrência e comportamento.

Na seção 4.1 são apresentados os artefatos da modelagem situacional. Em 4.2 é proposto um modelo de gerência de situações com base numa plataforma de regras abstrata.

4.1 INSPIRAÇÕES DA MODELAGEM DE SITUAÇÕES CONTEXTUAIS

Em (COSTA, 2007), uma situação contextual é definida como um estado particular de interesse acerca de um conjunto de entidades e seus respectivos contextos. Refere-se como *tipo de situação*, um grupo de situações contextuais que apresentam características similares. Por exemplo, o tipo de situação “*paciente diagnosticado com gripe*” denota todos os episódios em que algum indivíduo da entidade paciente foi caracterizado com gripe, ou seja, um *tipo de situação* representa instâncias de situação constituídas de indivíduos das mesmas entidades e tipos de contexto, relacionados sob as mesmas condições. A fundamentação ontológica de (COSTA, 2007) identifica duas categorias de contexto, denominadas: *contexto intrínseco* e *contexto relacional*, respectivamente. Os padrões situacionais são, em parte, derivações dessas duas espécies de contexto, portanto, defini-las é necessário.

Contexto intrínseco refere-se a contextos que pertencem a uma única entidade e não dependem de relacionamento com outras entidades. A localização de uma pessoa, seu humor ou sua temperatura são exemplos de contextos intrínsecos. Por outro lado, um *contexto relacional* é caracterizado por envolver duas ou mais entidades. A relação de amizade entre duas pessoas é um exemplo de contexto relacional. A Figura 4.1 exemplifica as duas categorias.

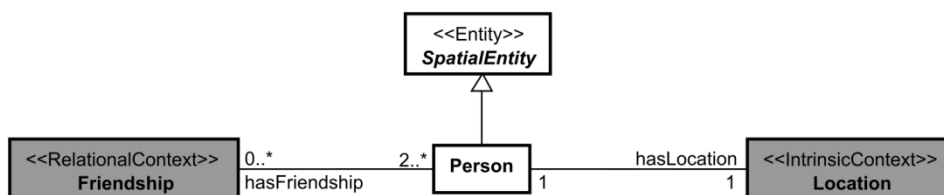


Figura 4.1 - Exemplos de Contexto Intrínseco e Relacional

Outro elemento importante para a modelagem de situações é o conceito de *relação formal*. Uma relação formal se estabelece entre indivíduos do mesmo tipo que apresentem qualidades comparáveis. No escopo dos modelos contextuais, elas se manifestam através de contextos cujo universo ofereça relações entre seus elementos. *Maior que*, *menor que*, *anterior a*, *posterior a*, *está contido em*, são exemplos genéricos de relações formais. Ainda na Figura 4.1, podemos supor relações formais envolvendo o contexto intrínseco *Location*, como, por exemplo, *mais próximo de* ou *mais distante de*.

No nível de categorização de tipos de situações, são identificados cinco padrões, são eles: *situação intrínseca*, *situação relacional*, *situação de relação formal*, *situação de situações* e *combinação de situações*. Situações intrínsecas envolvem apenas uma entidade e um contexto específico, sendo este intrínseco. Analogamente, situações relacionais são constituídas de duas ou mais entidades que compartilham um contexto relacional. Já situações de relação formal são compostas por ao menos dois indivíduos e algum de seus contextos intrínsecos, de forma que exista alguma relação formal entre os mesmos. Sobre estes três padrões primitivos são definidos os dois últimos. Uma situação de situações é uma composição sobre situações mais básicas, e uma combinação de situações é caracterizada por qualquer cenário que apresente uma permutação dos padrões situacionais anteriores.

A tipificação de situações contextuais é útil no aspecto estrutural de sua especificação. Na outra mão, o aspecto dinâmico de um tipo de situação envolve a descrição das condições impostas aos seus componentes contextuais para que a situação exista de fato. (COSTA, 2007) lança mão de especificações em OCL (*Object Constraint Language*) 2.0 a fim de cobrir esta visão.

4.2 MODELO DE MÁQUINA DE SITUAÇÕES

Entende-se como **tipo de situação** (*Situation Type*) um padrão que caracteriza um estado de interesse manifestável dentro do escopo de um determinado domínio de discurso. Caracterização esta, de natureza declarativa, que discursa sobre propriedades inerentes a, e relações (contextos) entre, entidades deste universo, restringindo-as. Em adição, diz-se, das entidades caracterizadas num **tipo de situação**, que as mesmas desempenham um **papel** na situação. Por exemplo, pode-se caracterizar o tipo de situação taquicardia declarando-a como um estado onde algum indivíduo apresenta frequência cardíaca elevada (acima de 100bpm). O indivíduo citado, sob as condições da situação, representa um taquicárdico, portanto, taquicárdico é um **papel situacional** do tipo de situação **taquicardia**.

O Modelo proposto neste capítulo supõe como base, um modelo abstrato de Sistema Baseado em Regras cuja arquitetura agrega características gerais de SBRs, como *working memory*, fatos, base de regras e máquina de inferência. Dessa forma é proposta uma especialização deste SBR abstrato, no sentido de suportar o conceito de *tipo de situação* introduzido previamente, de modo que, tal suporte avança sobre os aspectos estrutural, ou estático, e comportamental, ou dinâmico. O primeiro no sentido de formalizar a composição do domínio nos quais situações ocorrem, e o próprio artefato de situação. O segundo relaciona-se à detecção e controle de instâncias de situações.

4.2.1 Conceituação Estrutural

Na Figura 4.2, estabelece-se um modelo estrutural para a base de conhecimento, ou *working memory*, da plataforma de situações, de forma que, todo elemento incluído na base de conhecimento é, ou torna-se, invariavelmente um fato, ou, como o modelo define, *FactType*. Ser um objeto do tipo *FactType* denota que a máquina de inferência é ciente de sua existência e o considera no processo de casamento de padrões de regras.

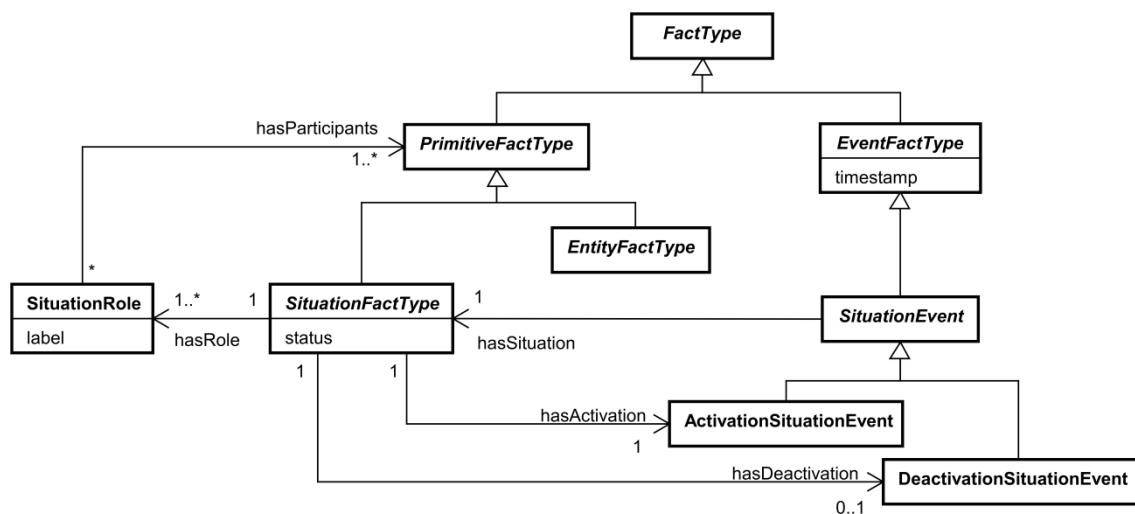


Figura 4.2 - Modelo Estrutural da Base de Conhecimento da Máquina de Situações

Descendo através da hierarquia do *FactType* nota-se a distinção entre fatos com características eventuais, *EventFactTypes*, que agrega métodos de correlação temporal (vide Figura 3.3), e os tipos restantes, classificados como *PrimitiveFactTypes*. Essa distinção é particularmente importante para expressar corretamente quais tipos de fatos podem participar de um *tipo de situação*, por sua vez, representado por *SituationFactType*, cuja composição é definida através de *SituationRoles*, os papéis situacionais. Que de maneira objetiva, significa dizer que apenas *EntityFactTypes* ou o próprio *SituationFactType* desempenham um papel na situação. Vale destacar que a temporalidade não é uma propriedade inerente de *SituationFactType*, mas sim, derivada de seus eventos delimitadores, na ativação (*ActivationSituationEvent*) e desativação (*DeactivationSituationEvent*).

Exemplificando, suponha o *tipo de situação* (*SituationFactType*) **febre**, que consiste de casos em que um indivíduo do tipo *Pessoa* apresenta temperatura maior que 37 graus. No escopo de **febre**, a entidade *Pessoa* assume uma participação bem definida na situação, que, por convenção, podemos chamar de *febril*, de modo que, *febril* é um tipo de papel que indivíduos do tipo *Pessoa* desempenham quando sob as premissas do tipo de situação **febre**. Um *papel situacional* (*SituationRole*) pode caracterizar não apenas um, mas uma pluralidade de indivíduos (1..*). Por exemplo, dado um tipo de situação **amizade**, caracterizada pela relação de amizade entre dois indivíduos distintos, ambos desempenham o mesmo papel, rotulado como *amigo*, no contexto da situação.

4.2.2 Modelo Comportamental de Situações

O aspecto comportamental de situações compreende a gerência do seu ciclo de vida. Resumidamente, o ciclo de vida de uma situação consiste da identificação de um tipo de situação, instanciação e manutenção até uma facultativa desativação, nesta ordem.

Chamamos de **instância de situação** a evidência do estabelecimento de um **tipo de situação** para um conjunto de indivíduos do domínio, durante um intervalo temporal contínuo. Destacam-se os extremos deste intervalo como: **instante de ativação**, quando a situação se apresenta, e de **desativação**, quando passa a não ser mais válida.

A instância, enquanto ativa, é única para um mesmo tipo X de situação e conjunto de relações (e_y, p_z) onde $e_y \in E$, o conjunto de entidades que desempenham algum papel na instância, e $p_z \in P$, conjunto de papéis definidos para o tipo de situação X .

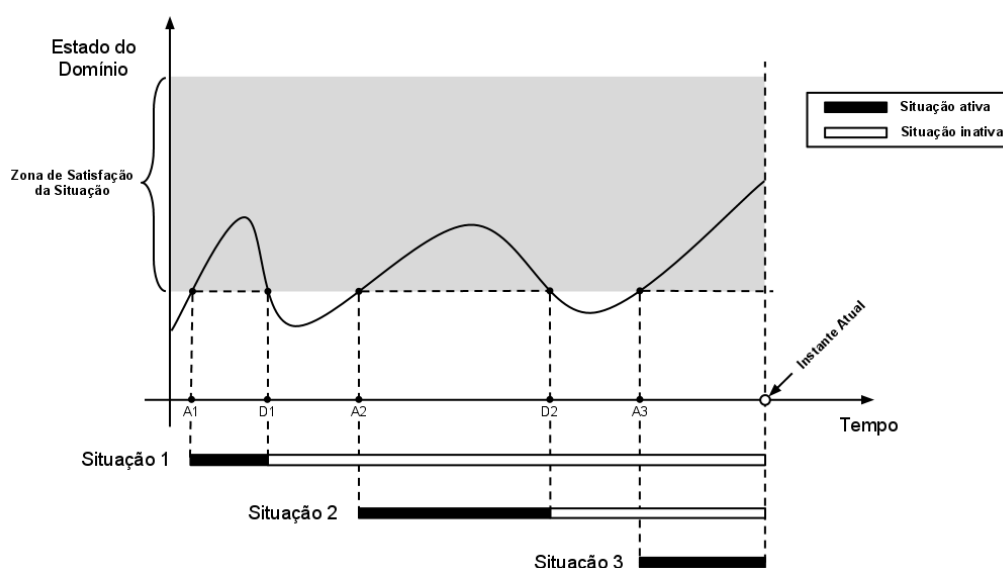


Figura 4.3 - Instâncias de um Tipo de Situação

A Figura 4.3 apresenta um gráfico cujos elementos do eixo Y representam combinações possíveis de estados das entidades de certo domínio, e onde suas coordenadas retratam o estado do domínio num determinado instante de tempo. Em favor da simplicidade, suponha, em Figura 4.3, um domínio limitado a apenas um atributo de um único indivíduo fixo (“José”), sua **frequência cardíaca**, por exemplo, e um tipo de situação, **taquicardia**, caracterizada quando a frequência do indivíduo atinge certo patamar (área cinza), tornando-o um **taquicardiaco**. Durante o intervalo retratado em Figura 4.3 ocorrem três instâncias de **taquicardia**. A unicidade de situações pode ser observada em Figura 4.3 quando *snapshots* do domínio em diferentes instantes são comparados, por exemplo, tome os instantes A_1 e A_x (onde $A_x < D_1$). Em A_1 não há instância prévia ativa de taquicardia, sendo assim taquicardia é instanciada para a tupla (**taquicardiaco**, “José”). Por sua vez, no instante A_x , o mesmo tipo de situação é válido para (**taquicardiaco**, “José”), mesma tupla instanciada em A_1 , e que se mantém ativa, portanto, a situação retratada no instante A_x é a mesma situação instanciada em A_1 .

Regras de Detecção

Baseando-se na proposição de regras para controlar as fases de uma situação, uma abordagem inicial seria definir um par de regras específicas para cada **tipo de situação (SituationFactType)**: uma responsável por sua ativação (instanciação) e outra por sua desativação. Dessa forma, numa descrição objetiva, a **ativação** de um

tipo de situação é dada quando as premissas são satisfeitas e não existe uma instância ativa do mesmo tipo de situação cujos participantes são os mesmos que já satisfizeram as estas premissas. Por outro lado, a **desativação** de uma situação se dá quando os participantes de uma situação ativa não satisfazem as premissas do tipo de situação instanciado. (COSTA, 2007) emprega essa abordagem (Figura 4.4).

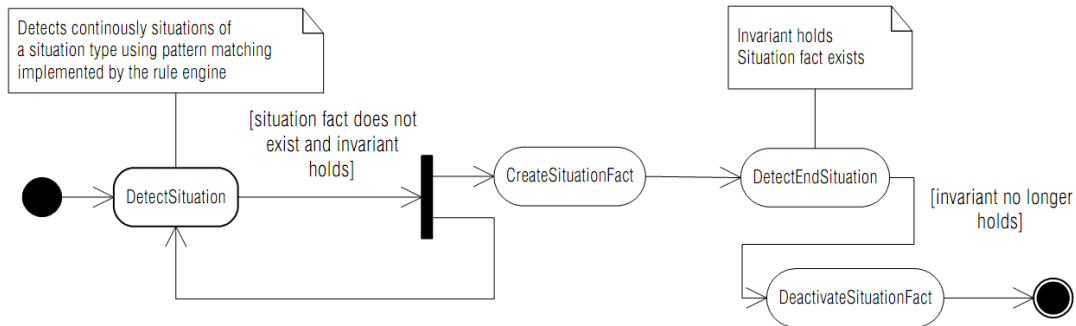


Figura 4.4 - Diagrama de Atividades para Gerência do Ciclo de Vida de Situações (COSTA, 2007)

Regra de Ativação

Representando a regra de ativação em lógica de predicados, temos a Proposição 3, composta pelas Proposições 1 e 2.

$$p: (constraint_1(i_1, r_1) \wedge \dots \wedge constraint_n(i_n, r_k))$$

Proposição 1 - Constraints da Situação

Na proposição 1, o predicado *constraint* representa uma restrição genérica aplicada sobre um indivíduo i_n , que, quando satisfeita, o caracteriza com um papel r_k . Um *tipo de situação* é composto por uma combinação finita desses predicados.

$$q: \exists y (situation(y) \wedge eq_type(x, y) \wedge active(y) \wedge (particip(y, i_1, r_1) \wedge \dots \wedge particip(y, i_n, r_k)))$$

Proposição 2 - Existência de Situação Ativa com Mesmos Participantes

Se a ativação de situações fosse baseada apenas na proposição 1, a cada instante no qual a mesma fosse avaliada como verdadeira, uma nova instância seria ativada, portanto, a proposição 2 é necessária para garantir a unicidade de instâncias enquanto sua ocorrência se estende durante o tempo. O predicado

particip avalia a presença de um indivíduo i_n compondo a situação y num papel r_k

$$\forall x(situation_type(x) \wedge p \wedge \sim q) \rightarrow create_situation(x)$$

Proposição 3 - Ativação de Situação

Aplicando essa regra para situação taquicardia, temos que, taquicardia é instanciada para um **indivíduo p_1 (do tipo *Paciente*) cuja frequência cardíaca exceda 100 bpm**, taquicardiaco, e não exista uma situação de Taquicardia, ativa, cujo indivíduo taquicardiaco seja p_1 .

Regra de Desativação

Analogamente à representação da *regra de ativação* temos a *regra de desativação* (Proposição 5), composta pelas proposições 1 e 4.

$$r: (situation(x) \wedge activ(x) \wedge (particip(x, i_1, r_1) \wedge \dots \wedge particip(x, i_n, r_k)))$$

Proposição 4 - Situação Ativa

$$\forall x(r \wedge \sim p) \rightarrow deactivate_situation(x)$$

Proposição 5 - Desativação de Situação

Desativar toda instância de taquicardia, ativa, cujo indivíduo i_1 , taquicardiaco, **não** apresente uma **frequência cardíaca acima de 100bpm**.

Esta abordagem apresenta inconvenientes à representação de situações. Primeiramente, à nível conceitual, um *tipo de situação* se baseia apenas em suas invariantes, que por sua vez, restringem o domínio de discurso, de modo que, é intuitivo estabelecer a ocorrência ou não de uma situação de acordo com o subdomínio resultante das restrições. Dessa maneira, a conceitualização de situações dispensa uma racionalização explícita sobre a desativação da situação, portanto, a especificação de situações, como proposta previamente, é contraintuitiva nesse sentido.

A verificação da pré-existência de instâncias ativas idênticas é potencialmente prejudicial à redigibilidade de regras de ativação de situações. Sem uma generalização adequada para validação de identidade entre instâncias de situação,





























suas respectivas regras podem ser poluídas por premissas irrelevantes, que se agravam proporcionalmente à complexidade do tipo de situação, no que se refere a quantidade de papéis situacionais.

A solução para nossa abordagem é desacoplar o ato de identificação de situações, da gerência do ciclo de vida de suas instâncias. Desse modo, tem-se duas camadas de regras. A mais adjacente consiste da regras de identificação de *tipos de situação*, cujas premissas seguem a linha da Proposição 1, as quais denominamos **Regras de Situação**. E a camada mais interna, que encapsula as premissas relacionadas a gerência de situação, sendo elas, respectivamente, **Regra de Ativação** e **Regra de Desativação**.

4.2.3 Relações Temporais de Situação

As relações temporais entre situações derivam da abstração da álgebra temporal entre eventos. Analogamente a elas, têm-se treze operadores temporais sobre situações. No entanto, devido à dinâmica dos *SituationFactTypes*, ora ativos, ora inativos, as correlações temporais sobre os mesmos, tem sua semântica afetada, dado que sua aplicabilidade é dependente do momento da avaliação. Por exemplo, supondo um par de situações **ativas**, *A* e *B*, respectivamente, avaliar se *A* ocorreu depois de *B* não é razoável, pois *B* nunca foi encerrada. Para os casos não aplicáveis, toda avaliação resulta em *falso*.

A Figura 4.5 retrata os treze tipos correlação considerando o contexto de possíveis estados das situações relacionadas.

	Active - Active	Active - Inactive	Inactive - Inactive
A Before B B After A		A  B 	A  B 
A Meets B B Met by A		A  B 	A  B 
A Overlaps B B Overlapped by A	A  B 	A  B 	A  B 
A Finishes B B Finished by A			A  B 
A Includes B B During A		A  B 	A  B 
A Starts B B Started by A	A  B 	A  B 	A  B 
A Coincides B			A  B 



 Active Situation
 Inactive Situation

Figura 4.5 - Relações Temporais entre Situações

5 DROOLS SITUATION

Neste capítulo será apresentada a realização do modelo discutido durante o capítulo anterior. Concepção do *Drools Situation*, as suas características e funcionamento.

A seção 5.1 apresenta a arquitetura global do *Drools Situation* a nível de usuário, a seção 5.2 discute o design de sua implementação, a seção 5.3 demonstra a aplicabilidade do *Drools Situation* para representar cenários situacionais propostos por (COSTA, 2007) . E em 5.4 têm-se as considerações finais do capítulo.

5.1 CONCEPÇÃO E ARQUITETURA GLOBAL

O *Drools Situation* consiste numa plataforma de apoio à especificação, detecção e gerência de *tipos de situação*, baseada no modelo de situações orientado a regras e composição de eventos, proposto neste trabalho. Este modelo representa uma especialização do modelo abstrato de sistemas baseados em regras clássicos. E, de forma análoga, no plano de realização, o *Drools Situation* representa a especialização de uma plataforma de regras, cujas situações constituem seu *first class citizen*.

5.1.1 JBoss Drools Como Plataforma: Motivação

Embora já apresentada em suas características gerais, no final do capítulo 3, vale ressaltar os fatores que determinaram a escolha do *JBoos Drools* como plataforma de realização.

Em primeiro lugar, modelo de eventos com lógica temporal completamente suportada e bem fundamentada, baseando-se nos trabalhos de (ALLEN, 1981), (ALLEN, 1983) e (BENNE, 2005). Junta-se a este fato a extensibilidade do Drools, cuja API (*Application Programming Interface*) permite a inclusão ou sobrecarga de

operadores, inclusive temporais, de modo que, uma adaptação da lógica temporal de eventos para uma lógica temporal de situações torna-se viável.

Outro fator é o **Truth Maintenance System**, componente do *Drools* responsável pelo controle de *inserções lógicas*, que consiste em atrelar a existência do fato lógico à validade das condições da regra que o produziu, desencadeando o recolhimento automático do fato. Este conceito é particularmente interessante, pois, se aproxima da abordagem para detecção do fim de uma situação, isto é, quando as invariantes da situação não são mais satisfeitas pelos componentes de uma instância.

A influência dos fatores acima destacados serão notadas na seção que discute a implementação do *Drools Situation*.

5.1.2 Arquitetura e Uso

A composição de uma aplicação baseada em *Drools Situations* apresenta um esquema semelhante ao retratado na Figura 5.1.

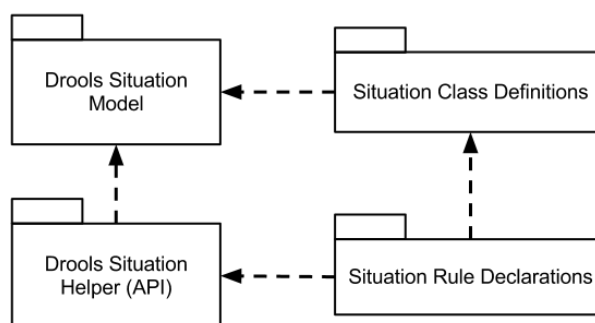


Figura 5.1 - Componentes de Aplicação Drools Situation

Um pacote com definições de tipos de situação (em *Java*), que estendem do modelo de situações (*Drools Situation Model*), muito semelhante ao especificado no diagrama da Figura 4.2, cujo principal elemento é *SituationType* (5-1) de modo que toda definição de situação especializa esta classe. Um pacote de declarações de regras de situação, em *DRL*. Em suas premissas referenciam as respectivas

definições de situação e na consequência utilizam a API do *Drools Situation*, que, por sua vez, é dependente do modelo.

```

1 public abstract class SituationType {
2
3     private Boolean          active;
4     private ActivateSituationEvent activation;
5     private DeactivateSituationEvent deactivation;
6
7     //GETTERS AND SETTERS
8 }

```

5-1 - Classe *SituationType*

5.1.3 Especificação de Situações

A especificação de tipos de situação em *Drools Situation* se dá através da declaração de dois artefatos: a *classe de situação* e sua respectiva *regra de situação*. *Classes de situação* são classes *Java* que estendem da classe abstrata *SituationType*, elemento central do *Drools Situation*, e descrevem seus papéis situacionais na forma de propriedades de classe. Por sua vez, as *regras de situação* aplicam as restrições, que caracterizam o tipo de situação, sobre o domínio, neste caso a *working memory* da sessão. Na sua composição são elencados (*casting*) os indivíduos de acordo com os papéis descritos na respectiva classe.

Suponha a especificação da situação de Febre, em 5-2, tem-se a declaração de sua *classe de situação* e em 5-3, sua regra.

```

1 public class Febre extends SituationType {
2     @SituationRole(label="febril")
3     private Paciente febril;
4     //GETTERS AND SETTERS
5 }

```

5-2- Classe de Situação

A correlação entre o tipo de situação e sua regra ocorre, de fato, por meio dos *SituationRoles*. Eles consistem de anotações (*Java Annotations*) sobre propriedades declaradas na classe de situação que indicam quais delas representam um papel a ser desempenhado no contexto da situação. No caso de 5-2, a propriedade **febril** foi anotada como um *SituationRole* cujo rótulo (*label*) corresponde à **febril**, dessa

forma, convencionou-se que a *regra de situação* deve atribuir, em sua LHS, como **\$febril**, um indivíduo que satisfaça as condições para tal, como representado em 5-3.

```

1 Rule "Febre"
2   when
3     $febril: Paciente(temperatura > 37)
4   then
5     SituationHelper.situationDetected(drools, kcontext, Febre.class)
6   end

```

5-3 - Regra de Situação

A *RHS* de uma regra de situação invoca a API do *Drools Situation*, resumidamente, o método *situationDetected*. Detalhes sobre a parametrização deste método serão abordados na sessão de implementação.

5.1.4 Multiplicidade de Papéis Situacionais

Por convenção, a especificação de papéis de multiplicidade maior que um é baseada em *Sets*.

```

1 public class Amizade extends SituationType {
2   @SituationRole(label="amigo")
3   private Set<Pessoa> amigos;
4   //GETTERS AND SETTERS
5 }

```

Figura 5.2 - Uso de Set para *SituationRole* "amigo"

No entanto, na LHS, cada indivíduo precisa ser atribuído separadamente. Nestes casos, o *Drools Situation* oferece um adendo à composição das *binding variables* na LHS, chamado indexador, cuja sintaxe é demonstrada em 5-4.

```

1 role$1 : Entity()
2 role$2 : Entity()
3 ...
4 role$<N>: Entity()

```

5-4 - *Role Bindings* com Index

A *engine* do *Situation* assume que os identificadores com *index* que apresentam o mesmo prefixo devem compor o mesmo *Set*. Além disso, a ordem de atribuição dos participantes é irrelevante dado que a ordem de elementos em um

conjunto também o é, o que significa dizer, por exemplo, que a instância da situação amizade, cuja regra de situação é declarada em 5-5, no qual o conjunto de amigos é dado por (I1, I2) é equivalente a uma instância para um conjunto de amigos (i2, i1).

```

1 Rule "Amizade"
2   when
3     amigo$1: Pessoa($amigos:amigos)
4     amigo$2: Pessoa() from $amigos
5   then
6     SituationHelper.situationDetected(drools, kcontext, Amizade.class)
end

```

5-5 - Regra de Situação "Amizade"

5.1.5 Raciocínio Temporal com Situações

Operadores

Os operadores temporais são e sua sintaxe é idêntica à empregada pelo *Drools Fusion* ao relacionar eventos.

```

1 Rule "FebreIntermitente"
2   when
3     $febreA: Febre(active==false)
4     $febreB: Febre(active==true, this before $febreA))
5   then
6     //action
7   end

```

5-6 - Aplicação de Operador Temporal sobre Situações

Vale ressaltar que os operadores temporais do *Drools Situation* são híbridos de modo que podem relacionar temporalmente situações e eventos.

Janelas

O *Drools Situation* não oferece um operador para definição de janelas específico para situações. No caso, é necessária referência explícita ao evento de situação, *ActivateSituationEvent* ou *DeactivateSituationEvent*.

```

1 Rule "CriseDeFebre"
2   when
3     $febre: Febre(active==true)
4     ActivateSituationEvent(situation==$febre) over window:time(1d)

```

5	then
6	//action
7	end

5-7 – Aplicação de Janela de Eventos de Situação

***KnowledgeSession* ciente de Situações**

Todos os recursos relativos ao *Drools Situation* não estão presentes numa *KnowledgeSession* padrão. É necessário a invocação de métodos para configuração de um *KnowledgeBuilder* para suportar o modelo, regras e operações sobre situações, que por sua vez, dará origem a uma *KnowledgeBase* e, posteriormente, a *KnowledgeSessions* cientes de situação.

O Módulo *SituationHelper* fornece três métodos para preparação do ambiente para o suporte à situações: o *setBuilderConfSituationAwareness*, o *setKnowledgeBuilderSituationAwareness* e o *setupRelativeSalience*, que devem ser aplicados sobre um *KnowledgeBuilderConfiguration*, e no caso dos dois últimos, sobre um *KnowledgeBuilderFactory*. O trecho de código em 5-8 retrata este processo.

1	<code>KnowledgeBuilderConfiguration builderConf;</code>
2	<code>KnowledgeBuilder kbuilder;</code>
3	
4	<code>builderConf = KnowledgeBuilderFactory.newKnowledgeBuilderConfiguration();</code>
5	<code>SituationHelper.setBuilderConfSituationAwareness(builderConf);</code>
6	<code>kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder(builderConf);</code>
7	<code>SituationHelper.setKnowledgeBuilderSituationAwareness(kbuilder);</code>
8	
9	<code>//<ADD ALL DRL RESOURCES HERE></code>
10	
11	<code>SituationHelper.setupRelativeSalience(kbuilder);</code>

5-8 - Configuração do *KnowledgeBuilder*

5.2 ARQUITETURA INTERNA E IMPLEMENTAÇÃO

A arquitetura base de conhecimento do *Drools Situation* é retratada pela Figura 5.3. Guiada pela abordagem de uma dupla camada de regras sugerida no capítulo anterior.

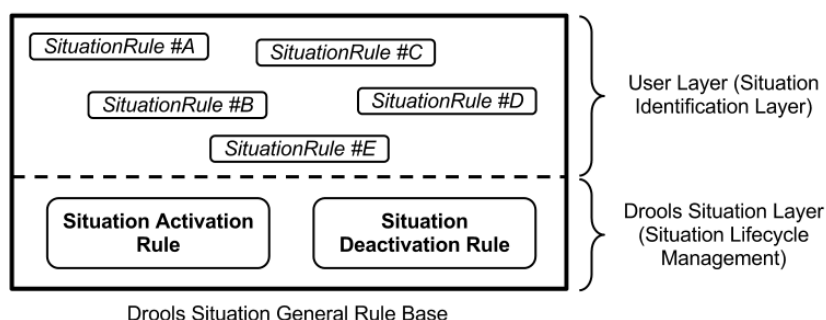


Figura 5.3 – Base de Regras do *Drools Situation*

O ciclo de vida de uma instância de situação consiste de sua criação, ativação, e desativação, sendo, esta última, facultativa, isto é, uma situação não necessariamente se encerra. Situações passadas, ou desativadas, são preservadas na *working memory* e podem vir a compor, como participante, tipos complexos de situações.

A fase ativa de uma situação corresponde ao intervalo no qual as premissas da respectiva regra de situação permanecem válidas para o mesmo contexto de participantes. Esta descrição é praticamente equivalente ao processo de inserção lógica de fatos, divergindo apenas no seu desfecho, de modo que, no primeiro ocorre uma mudança de estado, e neste último a exclusão do fato da *working memory*.

A estratégia adotada foi gerenciar o ciclo de vida de situações por meio de dois artefatos: a instância de situação em si, um *SituationType*, e um fato inserido logicamente, que denominamos *CurrentSituation*, cuja existência representa a fase ativa da respectiva instância de situação. A Figura 5.4 retrata o fragmento de modelo referente a classe *CurrentSituation*.

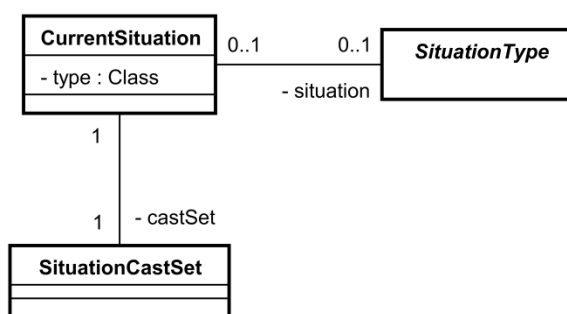


Figura 5.4 - CurrentSituation

A Classe *SituationCastSet* é uma especialização de uma estrutura *hash* que representa o elenco de uma situação relaciona os rótulos de um determinado papel, definido por meio de anotações Java nas classes de situação, com os respectivos objetos atribuídos durante a ativação da regra.

A distribuição deste processo entra as camadas de regras se dá da seguinte forma: a *camada de identificação de situações* lida com a instanciação de *CurrentSituations*. Por sua vez, a *camada de gerência de ciclo de vida* considera a existência com *SituationTypes*.

Por definição a RHS das regras de situação invocam a API do *SituationHelper* `situationDetected`, resumidamente, gera um *SituationCastSet* composto dos fatos capturados, na ativação, pelas *binding variables* da LHS. Por fim, um *CurrentSituation* é instanciado, tem atribuídos, seu `castSet` e `type` e é inserido logicamente na *working memory*.

Neste ponto, vale ressaltar uma característica da inserção lógica que consiste na não-instanciação de fatos equivalentes ou idênticos, de modo que, todo processo de inserção lógica realiza essa validação. O conceito de identidade de um fato é determinado pelo usuário, e, para nossa estratégia, o conceito de identidade definido para instâncias de *CurrentSituation* é equivalente ao discutido no Capítulo 4. Suponha dois momentos na detecção da situação *Febre*: no primeiro um indivíduo i_1 acaba de alcançar uma temperatura de 38 graus, e sob essas circunstâncias a *regra de situação* é disparada. Uma instância de *CurrentSituation* é criada e inserida logicamente na *working memory* e, então, i_1 torna-se um *febril*. Numa medição posterior, constata-se o aumento da temperatura do mesmo indivíduo, i_1 , agora com

39 graus, dessa forma a regra de situação é novamente disparada, uma *CurrentSituation* é instanciada, porém, a inserção lógica ignora esta nova instância, dado que ela é idêntica a, anterior, que continua ativa.

O *Drools Situation* possui um pacote DRL interno, onde mantém as declarações das regras de ativação e desativação de situação.

```

1 Package org.drools.situation.base
2
3 declare SituationEvent
4     @role(event)
5     @timestamp(timestamp)
6 end
7
8 rule "SituationActivation"
9     when
10         $act: CurrentSituation(situation == null,
11                               $type: type,
12                               $castset: castset)
13     then
14         $act.setSituation(SituationHelper.activateSituation(drools,
15                                                            $castset,
16                                                            $type)
17                           );
18         update($act);
19     end
20
21 rule "SituationDeactivation"
22     when
23         $sit: SituationType(active==true)
24         not (exists CurrentSituation(situation == $sit))
25     then
26         SituationHelper.deactivateSituation(drools, (Object) $sit);
27     end

```

5-9 - Regras de Gerência de Ciclo de Vida de Situações

Quando um *CurrentSituation* é inserido na *working memory*, seu atributo *situation* é propositalmente nulo, o que indica que sua respectiva instância de *SituationType* não foi criada.

A *Regra de Ativação* age como uma consumidora de *CurrentSituations*, instanciando seus respectivos *SituationTypes* de acordo com o tipo (*type*), e atribuindo corretamente o elenco agregado pelo *castSet*. Nesse momento um *ActiveSituationEvent* é registrado. Por sua vez, a *Regra de Desativação* busca por *SituationTypes* “órfãos”, isto é, que não possuem um *CurrentSituation* associado, devido ao recolhimento automático do mesmo, e quando disparada invoca

deactivateSituation, responsável por registrar o evento de desativação (*DeactivateSituationEvent*) e negar o status de ativo do *SituationType*.

O *SituationHelper* reúne todos os métodos que operam sobre a detecção e controle de instâncias de situação, visto que, toda regra relacionada à situação, seja para detecção, ativação ou desativação, invoca alguma operação do *Helper*. O código completo do *SituationHelper* é apresentado no Anexo B – Código do *SituationHelper*.

5.2.1 Sobrescrição de Operadores Temporais

A API do *Drools* oferece uma interface, denominada *EvaluatorDefinition*, para definição de novos operadores de regras, que por sua vez, são plugáveis à plataforma. A implementação do *Drools Situation* tira proveito deste recurso efetuando a sobrescrição de todos os operadores temporais originais do *Drools Fusion*, por novas implementações dos mesmos, de forma a suportarem *SituationType* como parâmetro.

Como indicado no modelo de situações proposto em 4, a racionalização temporal de situações, não é inerente a situações em si, mas, se dá sobre seus eventos delimitadores. No nível de implementação, a álgebra temporal continua a ser propriedade exclusiva de eventos. Em avaliações onde algum parâmetro é um *SituationType*, o novo operador apenas necessita efetuar a extração do evento de situação que interessa ao caso e aplicar a lógica temporal de acordo com a implementação antiga. Vale ressaltar que a nova implementação dos operadores é híbrida, e portanto, capaz de lidar com uma correlação temporal de um tipo de situação e um evento, e não apenas de parâmetros homogêneos.

5.3 PADRÕES DE SITUAÇÕES EM *DROOLS SITUATION*

Como abordado brevemente no início do capítulo 4, (COSTA, 2007) identificou cinco padrões situacionais possíveis em seu modelo, sendo eles: *situação*

intrínseca, situação relacional, situação de relação formal, situação de situação e combinação de situações, respectivamente. Para cada tipo citado, (COSTA, 2007) apresenta cenários. Nesta seção, alguns destes cenários serão instanciados para a plataforma *Drools Situation* a fim de demonstrar, que apesar de independentes, a plataforma *Situation* é capaz de representar os padrões de (COSTA, 2007). Em favor da simplicidade, os exemplos a seguir dão ênfase às definições das *regras de situação* em comparação às definições de situação em *OCL*.

5.3.1 Situação Intrínseca

(COSTA, 2007) sugere um cenário de situação intrínseca que consiste nos casos em que um indivíduo, usuário do **MSN** e **Skype**, encontra-se disponível em algum desses serviços. A Figura 5.5 retrata o modelo de contexto do cenário.

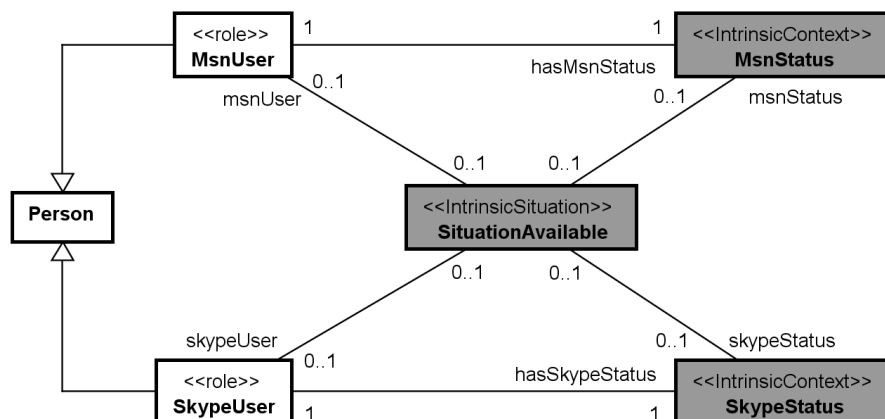


Figura 5.5 - Modelo da Situação Intrínseca *SituationAvailable* (COSTA, 2007)

Em 5-10 têm-se a declaração da *SituationAvailable* em OCL. E por sua vez, temos em 5-11, a sua equivalente em *Regra de Situação*.

```

1  {Context SituationAvailable inv:
2  (skypeUser = msnUser) AND
3  ((not skypeUser.oclIsUndefined()) AND
4  (skypeUser.skypeStatus = skypeStatus) AND
5  ((skypeStatus.value = "Online")
6   OR (skypeStatus.value = "SkypeMe"))))
7  OR
8  ((not msnUser.oclIsUndefined()) AND
9  (msnUser.msnStatus = msnStatus) AND
10 ((msnStatus.value = "Online")

```

11	<code>OR (msnStatus.value = "BeRightBack"))))}</code>
----	---

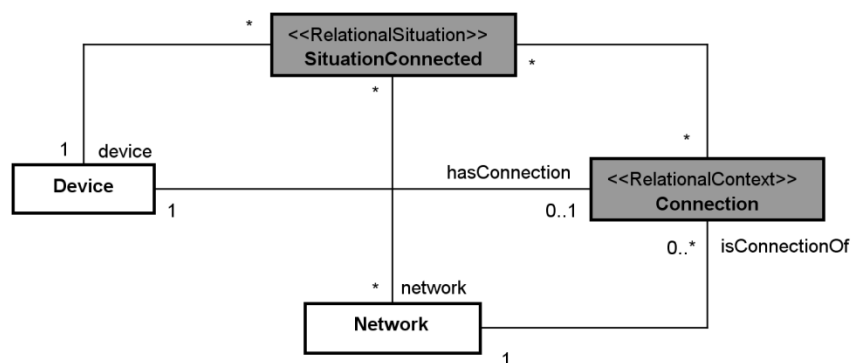
5-10 - Definição OCL para *SituationAvailable* (COSTA, 2007)

1	Rule "SituationAvailable"
2	when
3	<code>\$MSNUser: MSNUser(\$MSNStatus: MSNStatus)</code>
4	<code>\$SkypeUser: SkypeUser(this==\$MSNUser, \$SkypeStatus: SkypeStatus)</code>
5	<code>(MSNStatus(this==\$MSNStatus, value(=="OnLine" =="BeRightBack")) or</code>
6	<code>SkypeStatus(this==\$SkypeStatus, value(=="OnLine" =="SkypeMe")))</code>
7	then
8	<code>SituationHelper.situationDetected(drools,</code>
9	<code>kcontext,</code>
	<code>SituationAvailable.class)</code>
	end

5-11 – Regra de Situação *SituationAvailable*

5.3.2 Situação Relacional

Para um cenário de situação relacional, (COSTA, 2007) propõe a situação onde um dispositivo (*Device*) apresenta uma conexão (*Connection*) estabelecida com alguma rede (*Network*). A Figura 5.6 apresenta o modelo do cenário.

Figura 5.6 - Modelo da Situação Relacional *SituationConnected* (COSTA, 2007)

Em 5-12 têm-se a declaração da *SituationConnected* por meio de OCL. E por sua vez, temos 5-13, a sua equivalente em *Regra de Situação*.

1	<code>{Context SituationConnected inv:</code>
2	<code>not device.hasConnection.oclIsUndefined() }</code>

5-12 - Definição OCL para *SituationConnected* (COSTA, 2007)

```

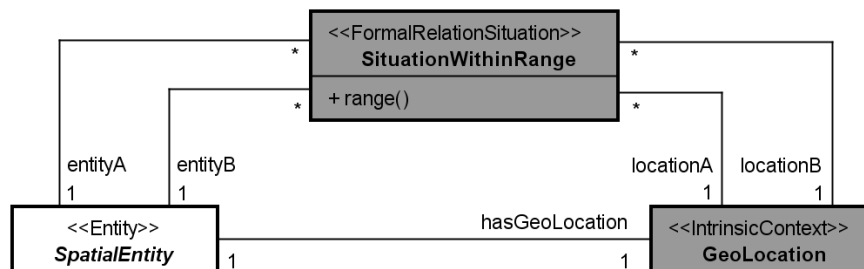
1 Rule "SituationConnected"
2   when
3     $network: Network()
4     $connection: Connection(network==$network)
5     $device: Device(connection==$connection)
6   then
7     SituationHelper.situationDetected(drools,
8                                       kcontext,
9                                       SituationConnected.class)
end

```

5-13- Regra "SituationConnected"

5.3.3 Situação de Relação Formal

O cenário proposto por (COSTA, 2007) para representar as situações de relação formal, foi o de proximidade entre entidades de um espaço.

Figura 5.7 - Modelo da Situação de Relação Formal *SituationWithinRange* (COSTA, 2007)

Em 5-14 têm-se a declaração da *SituationConnected* por meio de OCL. E por sua vez, temos 5-15, a sua equivalente em *Regra de Situação*.

```

1 {Context SituationWithinRange inv:
2   entityA.hasGeoLocation = locationA AND
3   entityB.hasGeoLocation = locationB AND
4   locationA.value->distance(locationB.value) < range}

```

5-14 - Definição OCL para *SituationWithinRange* (COSTA, 2007)

```

1 Rule "SituationWithinRange"
2   when
3     $entityA: SpatialEntity($locationA: location)
4     $entityB: SpatialEntity($locationB: location)
5     eval($locationA.value.distance($locationB) < range)
6   then
7     SituationHelper.situationDetected(drools,
8                                       kcontext,

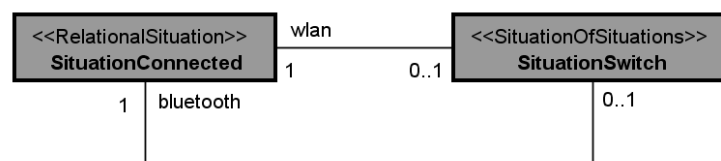
```

9	<code>SituationWithinRange.class)</code>
10	<code>end</code>

5-15 - Regra da Situação *SituationWithinRange*

5.3.4 Situação de Situações

Na Figura 5.8 tem-se o modelo de uma situação *SituationSwitch*, a qual se estabelece quando há um determinado dispositivo abandona uma rede *WLAN* e se associa a uma rede *Bluetooth*.

Figura 5.8- Modelo da Situação de Situações *SituationSwitch* (COSTA, 2007)

```

1 { Context SituationSwitch inv:
2   (wlan.device = bluetooth.device) AND
3   (wlan.device.hasConnection.network.oclIsTypeOf(WLAN)) AND
4   (bluetooth.device.hasConnection.network.oclIsTypeOf(Bluetooth))
5   AND (bluetooth.initialtime - wlan.finaltime < 1)}

```

5-16 - Definição OCL para *SituationSwitch* (COSTA, 2007)

Em 5-16 têm-se a declaração da *SituationSwitch* por meio de OCL. E por sua vez, temos 5-175-15, a sua equivalente em *Regra de Situação*.

```

1 rule "SituationSwitch"
2   when
3     $wlan: SituationConnected($device: device,
4                               network.type==WLAN)
5     $bluetooth: SituationConnected(device==$device,
6                                    network.type==BLUETOOTH,
7                                    this metby[1s] $wlan)
8   then
9     SituationHelper.situationDetected(drools,
10                                       kcontext,
11                                       SituationSwitch.class)
12 end

```

5-17 - Regra da Situação *SituationSwitch*

5.3.5 Combinação de Situações

A Combinação de Situações é uma padrão que envolve dois o mais padrões de situação, como os mostrados anteriormente. O cenário proposto na

Figura 5.9 é o de uma *Situação de Apresentação* (*SituationPresentation*) que se estabelece enquanto um dispositivo for capaz de se manter ligado tempo suficiente para que o *download* de uma apresentação seja efetuado, e que a mesma possa ser apresentada.

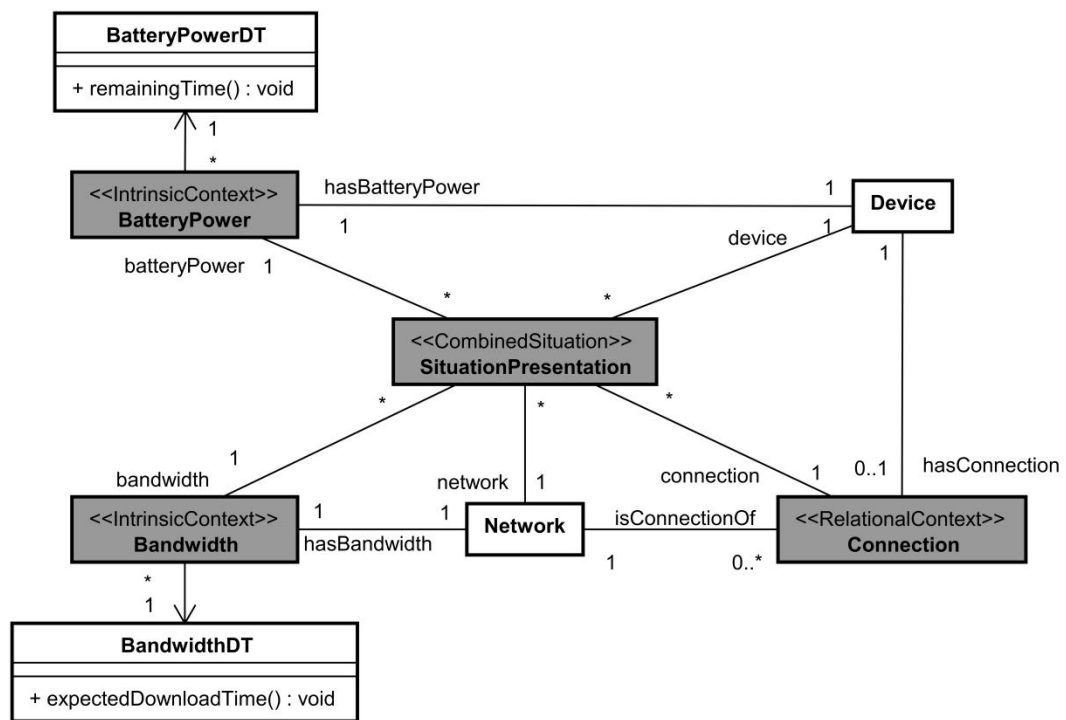


Figura 5.9 - Modelo de Situação Combinada *SituationPresentation* (COSTA, 2007)

Em 5-18 têm-se a declaração da *SituationPresentation* por meio de OCL. E por sua vez, temos 5-195-15, a sua equivalente em *Regra de Situação*.

1	{ Context SituationPresentation inv:
2	(not device.hasConnection.isOclUndefined()) AND
3	(device.hasConnection = connection) AND
4	(connection.network = network) AND
5	(network.hasBandwidth = bandwidth) AND
6	(device.hasBatteryPower = batterypower) AND

7	((<i>bandwidth</i> .value->expectedDownloadTime(<i>presentationsize</i>) +
8	<i>presentationduration</i>) < <i>batterypower</i> .value->remainingTime()))}

5-18 – Definição OCL para *SituationPresentation* (COSTA, 2007)

1	rule "SituationPresentation"
2	when
3	<i>\$network</i> : Network(<i>\$bandwidth</i> : bandwidth)
4	<i>\$connection</i> : Connection(network== <i>\$network</i>)
5	<i>\$device</i> : Device(connection== <i>\$connection</i> ,
6	<i>\$batterypower</i> : batterypower)
7	eval((<i>\$bandwidth</i> .value.expectedDownloadTime(<i>presentationsize</i>) +
8	<i>presentationduration</i>) < <i>\$batterypower</i> .value.remainingTime)
9	then
10	<i>SituationHelper</i> .situationDetected(drools,
11	kcontext,
12	<i>SituationPresentation</i> .class)
13	end

5-19 - Regra de Situação *SituationPresentation*

5.4 CONSIDERAÇÕES

É importante salientar que a abordagem que guiou o desenvolvimento deste módulo tinha como requisito chave a redigibilidade de tipo de situações. As decisões de projeto decorrentes dessa linha, a destacar a própria arquitetura da dupla camada de regras, podem ter um efeito negativo sobre a performance do sistema. No entanto, este possível impacto no desempenho não foi mensurado durante a o processo de implementação.

No que se refere à expressividade da ferramenta, esta foi bem sucedida em todos os cenários situacionais propostos, sendo, em alguns casos mais completa ou adequada, principalmente quando exigido raciocínio temporal. Vide o cenário da situação *SituationSwitch* em 5.3.4.

6 CONCLUSÃO E TRABALHOS FUTUROS

A partir de um estudo sobre plataformas de regras, este trabalho propôs um modelo de execução para detecção de situações e o aplicou no desenvolvimento de uma plataforma de situações. Por sua vez, a ferramenta desenvolvida permite especificar estruturalmente e dinamicamente tipos de situações, sendo que esta última se dá de maneira clara apenas descrevendo as invariantes que caracterizam a situação, e na sua execução abstrai toda a gerência do ciclo de vida de situações. Em adição, a plataforma também dá suporte à correlação temporal entre pares *situação-situação* e *situação-evento*.

Foi demonstrado que a ferramenta é capaz de expressar as situações contextuais de (COSTA, 2007), tais como situações intrínsecas, relacionais, de relação formal, de situações e combinadas. Este fato dá margem a uma futura interação entre os dois modelos.

6.1 TRABALHOS FUTUROS

O modelo de situações de (COSTA, 2007) preserva *snapshots* de estado dos participantes da situação no momento de sua desativação, esta estratégia é particularmente útil para a composição de situações de situações. O *Drools Situation* não avançou nessa funcionalidade, o que pode causar ao expressar esse padrão, no entanto, seria recomendável que fosse implementada como trabalho futuro.

Como abordado nas considerações do capítulo anterior, durante o processo de desenvolvimento do *Drools Situation* não houve uma estimativa do impacto da arquitetura sobre a performance da *engine*. Uma linha de trabalho futuro é a de analisar as decisões arquiteturais sob a luz da performance, utilizando cenários de situação robustos, que se aproximem das reais, possivelmente na área de detecção de fraudes bancárias, domínio que já esteve na pauta deste trabalho.

As alterações promovidas pela plataforma *Drools Situation*, tornaram o editor de DRL um tanto inadequado em alguns casos, por exemplo, nas declarações de

operadores temporais entre situações, para os quais o analisador sintático não reconhece como válidas. Alterações no analisador poderiam ser aplicadas a fim de suportar plenamente essa nova sintaxe.

REFERÊNCIAS

DEY, A. K.; ABOWD, G. D. Towards a Better Understanding of Context and Context-Awareness. 1999.

COSTA, D. P.; GUIZZARDI, G.; ALMEIDA, J.P.; PIRES, F. L., SINDEREN, M., Situations in Conceptual Modeling of Context. **Workshop on Vocabularies, Ontologies, and Rules for the Enterprise**, 2006.

COSTA, D. P.; GUIZZARDI, G.; ALMEIDA, J.P.; PIRES, F. L.; SINDEREN, M., Situation Specification and Realization in Rule-Based Context-Aware Applications. **International Conference on Distributed Applications and Interoperable Systems**, 2007.

DEY, A. K. Understanding and Using Context. **Personal and Ubiquitous Computing**, p. 4-7, 2001.

ADI, A. A Language and an Execution Model for the Detection of Active Situations, 2002.

GOIX, L.; VALLA, M.; CERAMI, L.; FALCARIN, P. Situation Inference for Mobile Users: a Rule Based Approach. **International Conference on Mobile Data Management**, p. 299-303, 2007.

FORGY, C. A Network Match Routine for Production Systems, **Working Paper**, 1974.

ETZION, O.; NIBBLET, P. **Entering the World of Event Processing**. Em: ETZION, O.; NIBBLET, P. *Event Processing in Action*. Greenwich: Manning, 2010. p. 3-29.

COSTA, P. D. **Architectural Support for Context-Aware Applications - From Context Models to Services Platforms**. University of Twente. Enschede, The Netherlands. 2007. PhD thesis.

YE, J.; DOBSON, S.; MCKEEVER, S., Situation Identification Techniques in Pervasive Computing: A Review. **Pervasive and Mobile Computing**, 2011.

GUIZZARDI, G. **Ontological Foundations for Structural Conceptual Models**. Centre for Telematics and Information Technology, University of Twente. Enschede, The Netherlands. 2005. PhD thesis.

SASIKUMAR, M. **Rule Based Systems**. Em: SASIKUMAR, M. *A Practical Introduction to Rule Based Expert Systems*. New Delhi: Narosa Publishing House, 2007. p. 21-41.

FRIEDMAN-HILL, E. **Introducing Rule Based Systems**. Em: FRIEDMAN-HILL, E. *Jess in Action: Rule-Based Systems in Java*. Greenwich: Manning, 2003. p. 3-27.

BALI, M. **Introduction**, Em: BALI, M. *Drools JBoss Rules 5.0 Developer's Guide*. Birmingham: Packt Publishing, 2009. p. 1-17.

DROOLS EXPERT. JBoss Drools 5.2 - Expert Documentation, 2011, Disponível em: <<http://www.jboss.org/drools/documentation/>>. Acesso em: Março 2012.

DROOLS FUSION. JBoss Drools 5.2 - Fusion Documentation, 2011, Disponível em: <<http://www.jboss.org/drools/documentation/>>. Acesso em: Março 2012.

MICHELSON M. Brenda. **Event-driven Architecture Overview**. Patricia Seybold Group, 2006.

SALATINO, M. **Emergency Services: Process, Rules and Events**. Plugtree, 2011.

ANEXO A – EXPRESSÕES DOS OPERADORES TEMPORAIS DE EVENTOS

Este anexo apresenta os operadores temporais suportados pelo *Drools Fusion*, bem como suas respectivas expressões algébricas.

After

LHS	<code>\$eventA : EventA(this after[3m30s, 4m] \$eventB)</code>
Expressão	<code>3m30s <= \$eventA.startTimestamp - \$eventB.endTimeStamp <= 4m</code>

Before

LHS	<code>\$eventA : EventA(this before[3m30s, 4m] \$eventB)</code>
Expressão	<code>3m30s <= \$eventB.startTimestamp - \$eventA.endTimeStamp <= 4m</code>

Coincides

LHS	<code>\$eventA : EventA(this coincides \$eventB)</code>
Expressão	<code>\$eventA.startTimestamp == \$eventB.startTimestamp && \$eventA.endTimeStamp == \$eventB.endTimeStamp</code>

During

LHS	<code>\$eventA : EventA(this during \$eventB)</code>
Expressão	<code>\$eventB.startTimestamp < \$eventA.startTimestamp <= \$eventA.endTimeStamp < \$eventB.endTimeStamp</code>

Finishes

LHS	<code>\$eventA : EventA(this finishes \$eventB)</code>
-----	--

Expressão	<code>\$eventB.startTimestamp < \$eventA.startTimestamp && \$eventA.endTimestamp == \$eventB.endTimestamp</code>
-----------	---

Finished By

LHS	<code>\$eventA : EventA(this finishes \$eventB)</code>
Expressão	<code>\$eventB.startTimestamp < \$eventA.startTimestamp && \$eventA.endTimestamp == \$eventB.endTimestamp</code>

Includes

LHS	<code>\$eventA : EventA(this includes \$eventB)</code>
Expressão	<code>\$eventA.startTimestamp < \$eventB.startTimestamp <= \$eventB.endTimestamp < \$eventA.endTimestamp</code>

Meets

LHS	<code>\$eventA : EventA(this meets \$eventB)</code>
Expressão	<code>abs(\$eventB.startTimestamp - \$eventA.endTimestamp) == 0</code>

Met By

LHS	<code>\$eventA : EventA(this metby \$eventB)</code>
Expressão	<code>abs(\$eventA.startTimestamp - \$eventB.endTimestamp) == 0</code>

Overlaps

LHS	<code>\$eventA : EventA(this overlaps \$eventB)</code>
Expressão	<code>\$eventA.startTimestamp < \$eventB.startTimestamp < \$eventA.endTimestamp < \$eventB.endTimestamp</code>

Overlapped By

LHS	<code>\$eventA : EventA(this overlappedby \$eventB)</code>
Expressão	<code>\$eventB.startTimestamp < \$eventA.startTimestamp < \$eventB.endTimestamp < \$eventA.endTimestamp</code>

Starts

LHS	<code>\$eventA : EventA(this starts \$eventB)</code>
Expressão	<code>\$eventA.startTimestamp == \$eventB.startTimestamp && \$eventA.endTimestamp < \$eventB.endTimestamp</code>

Started By

LHS	<code>\$eventA : EventA(this startedby \$eventB)</code>
Expressão	<code>\$eventA.startTimestamp == \$eventB.startTimestamp && \$eventA.endTimestamp > \$eventB.endTimestamp</code>

ANEXO B – CÓDIGO DO *SITUATIONHELPER*

Este anexo contém o código da classe *SituationHelper*, que compõe a plataforma *Drools Situation*, apresentada no Capítulo 5.

```

1 package org.drools.situation.base;
2
3 import java.lang.annotation.Annotation;
4 import java.lang.reflect.Field;
5 import java.util.ArrayList;
6 import java.util.Collection;
7 import java.util.HashSet;
8 import java.util.LinkedList;
9 import java.util.List;
10 import java.util.Set;
11
12 import org.drools.definition.KnowledgePackage;
13 import org.drools.definitions.rule.impl.RuleImpl;
14 import org.drools.base.SalienceInteger;
15 import org.drools.builder.KnowledgeBuilder;
16 import org.drools.builder.KnowledgeBuilderConfiguration;
17 import org.drools.builder.ResourceType;
18 import org.drools.builder.conf.EvaluatorOption;
19 import org.drools.io.ResourceFactory;
20 import org.drools.rule.Rule;
21 import org.drools.runtime.rule.Activation;
22 import org.drools.runtime.rule.RuleContext;
23 import org.drools.spi.KnowledgeHelper;
24
25 import org.drools.situation.base.evaluators.AfterEvaluatorDefinition;
26 import org.drools.situation.base.evaluators.BeforeEvaluatorDefinition;
27 import org.drools.situation.base.evaluators.FinishesEvaluatorDefinition;
28 import org.drools.situation.base.evaluators.FinishedByEvaluatorDefinition;
29 import org.drools.situation.base.evaluators.OverlapsEvaluatorDefinition;
30 import org.drools.situation.base.evaluators.OverlappedByEvaluatorDefinition;
31 import org.drools.situation.base.evaluators.StartsEvaluatorDefinition;
32 import org.drools.situation.base.evaluators.StartedByEvaluatorDefinition;
33 import org.drools.situation.base.evaluators.MeetsEvaluatorDefinition;
34 import org.drools.situation.base.evaluators.MetByEvaluatorDefinition;
35 import org.drools.situation.base.evaluators.includesEvaluatorDefinition;
36 import org.drools.situation.base.evaluators.DuringEvaluatorDefinition;
37 import org.drools.situation.base.evaluators.CoincidesEvaluatorDefinition;
38
39 // @SuppressWarnings("restriction")
40 public class SituationHelper {
41
42     public static void setupRelativeSalience(KnowledgeBuilder kbuilder) {
43
44         KnowledgePackage situationBasePkg = null;
45         Integer maxSalience = null;
46
47         for (KnowledgePackage pkg: kbuilder.getKnowledgePackages()) {
48
49             if (pkg.getName().equals("org.drools.situation.base")) situationBasePkg = pkg;
50
51             for (Rule rule: getRulesFromPackage(pkg)) {
52
53                 if (maxSalience != null) {
54                     if (rule.getSalience().getValue(null, null, null) > maxSalience) {
55                         maxSalience = new Integer(rule.getSalience().getValue(null, null, null));
56                     }
57                 }
58             }
59         }
60     }
61 }

```

```

56     }
57     } else maxSaliency = new Integer(rule.getSaliency().getValue(null, null, null));
58
59     }
60 }
61
62 if (situationBasePkg != null) {
63     getRuleFromPackage(situationBasePkg,
64         "SituationActivation").setSaliency(new
65             SaliencyInteger(maxSaliency + 1));
66     getRuleFromPackage(situationBasePkg,
67         "SituationDeactivation").setSaliency(new
68             SaliencyInteger(maxSaliency + 2));
69 }
70 }
71
72 private static Collection<Rule> getRulesFromPackage(KnowledgePackage pkg) {
73
74     LinkedList<Rule> collection = new LinkedList<Rule>();
75
76     for (Object obj: pkg.getRules()) {
77
78         RuleImpl ruleImpl = (RuleImpl) obj;
79
80         Field ruleField;
81         try {
82             ruleField = ruleImpl.getClass().getDeclaredField("rule");
83
84             ruleField.setAccessible(true);
85             Rule rule = (Rule) ruleField.get(ruleImpl);
86             ruleField.setAccessible(false);
87             collection.add(rule);
88
89         } catch (SecurityException e) {
90             // TODO Auto-generated catch block
91             e.printStackTrace();
92         } catch (NoSuchFieldException e) {
93             // TODO Auto-generated catch block
94             e.printStackTrace();
95         } catch (IllegalArgumentException e) {
96             // TODO Auto-generated catch block
97             e.printStackTrace();
98         } catch (IllegalAccessException e) {
99             // TODO Auto-generated catch block
100             e.printStackTrace();
101         }
102     }
103     return collection;
104 }
105
106 private static Rule getRuleFromPackage(KnowledgePackage pkg, String rulename) {
107
108     Field ruleField;
109
110     for (Object obj: pkg.getRules()) {
111
112         RuleImpl ruleImpl = (RuleImpl) obj;
113
114         if (ruleImpl.getName().equals(rulename)) {
115             try {
116                 ruleField = ruleImpl.getClass().getDeclaredField("rule");
117                 ruleField.setAccessible(true);
118                 Rule rule = (Rule) ruleField.get(ruleImpl);
119                 ruleField.setAccessible(false);
120                 return rule;
121             } catch (SecurityException e) {
122                 // TODO Auto-generated catch block

```

```

123         e.printStackTrace();
124     } catch (NoSuchFieldException e) {
125         // TODO Auto-generated catch block
126         e.printStackTrace();
127     } catch (IllegalArgumentException e) {
128         // TODO Auto-generated catch block
129         e.printStackTrace();
130     } catch (IllegalAccessException e) {
131         // TODO Auto-generated catch block
132         e.printStackTrace();
133     }
134 }
135 }
136 return null;
137 }
138
139
140 public static void setBuilderConfSituationAwareness(
141     KnowledgeBuilderConfiguration builderConf) {
142
143     builderConf.setOption(EvaluatorOption.get("after",
144         new AfterEvaluatorDefinition()));
145     builderConf.setOption(EvaluatorOption.get("before",
146         new BeforeEvaluatorDefinition()));
147     builderConf.setOption(EvaluatorOption.get("overlaps",
148         new OverlapsEvaluatorDefinition()));
149     builderConf.setOption(EvaluatorOption.get("overlappedby",
150         new OverlappedByEvaluatorDefinition()));
151     builderConf.setOption(EvaluatorOption.get("finishes",
152         new FinishesEvaluatorDefinition()));
153     builderConf.setOption(EvaluatorOption.get("finishedby",
154         new FinishedByEvaluatorDefinition()));
155     builderConf.setOption(EvaluatorOption.get("starts",
156         new StartsEvaluatorDefinition()));
157     builderConf.setOption(EvaluatorOption.get("startedby",
158         new StartedByEvaluatorDefinition()));
159     builderConf.setOption(EvaluatorOption.get("meets",
160         new MeetsEvaluatorDefinition()));
161     builderConf.setOption(EvaluatorOption.get("metby",
162         new MetByEvaluatorDefinition()));
163     builderConf.setOption(EvaluatorOption.get("includes",
164         new IncludesEvaluatorDefinition()));
165     builderConf.setOption(EvaluatorOption.get("during",
166         new DuringEvaluatorDefinition()));
167     builderConf.setOption(EvaluatorOption.get("coincides",
168         new CoincidesEvaluatorDefinition()));
169 }
170
171
172 public static void setKnowledgeBuilderSituationAwareness(KnowledgeBuilder kbuilder) {
173     kbuilder.add(ResourceFactory.newClassPathResource(
174         "org/drools/situation/base/SituationBaseRules.drl"),
175         ResourceType.DRL);
176 }
177
178 public static List<Field> GetSituationRoleFields(Class<?> sit) {
179
180     List<Field> situationRoleFields = new LinkedList<Field>();
181
182     //recursively gets fields on superclass hierarchy
183     if (sit.getSuperclass() != null) {
184         situationRoleFields.addAll(GetSituationRoleFields(sit.getSuperclass()));
185     }
186
187     Field[] fields = sit.getDeclaredFields();
188     for(Field field: fields) {
189         Annotation[] annotations = field.getDeclaredAnnotations();

```

```

190     for (Annotation annotation: annotations) {
191         if (annotation instanceof SituationRole) {
192             situationRoleFields.add(field);
193         }
194     }
195 }
196 return situationRoleFields;
197 }
198
199 public static SituationCastSet getSituationCast(Activation activation,
200                                             Class<?> type) throws Exception {
201
202     int counter;
203     String roleLabel;
204     Object obj;
205     SituationCastSet castset = new SituationCastSet();
206     Rule rule = (Rule) activation.getRule();
207     List<String> LHSIdentifiers = new ArrayList<String>(rule.getDeclarations().keySet());
208     List<Field> situationRoleFields = GetSituationRoleFields(type);
209
210     for(Field field: situationRoleFields) {
211         if (field.getType().equals(java.util.Set.class)) {
212             Set<Object> set = new HashSet<Object>();
213             counter = 1;
214             do {
215                 obj = null;
216                 roleLabel = new String(field.getName() + "$" + counter);
217                 if (LHSIdentifiers.contains(roleLabel)) {
218                     obj = activation.getDeclarationValue(roleLabel);
219                     set.add(obj);
220                 }
221                 counter++;
222             } while (obj != null);
223             if (set.size() < 2) throw new Exception();
224             castset.put(field.getName(), set);
225         }
226         else {
227             castset.put(field.getName(),
228                         activation.getDeclarationValue(field.getName()));
229         }
230     }
231     return castset;
232 }
233
234 public static void SetFieldsFromMatchedObjects(Object obj,
235                                             SituationCastSet sourceRuleObjs) {
236
237     List<Field> targetObjFields = GetSituationRoleFields(obj.getClass());
238
239     for(Field field: targetObjFields) {
240         Object declaredObj = sourceRuleObjs.get(field.getName());
241         if (declaredObj != null) {
242             try {
243                 field.setAccessible(true);
244                 field.set(obj, declaredObj);
245                 field.setAccessible(false);
246             } catch (IllegalArgumentException e) {
247                 e.printStackTrace();
248             } catch (IllegalAccessException e) {
249                 e.printStackTrace();
250             }
251         }
252     }
253 }
254
255 public static void situationDetected(KnowledgeHelper khelper,
256

```

```

RuleContext kcontext,
Class<?> type) throws Exception {

    CurrentSituation asf = new CurrentSituation(type);
    asf.setTimestamp(khelper.getKnowledgeRuntime().getSessionClock().getCurrentTime());
    asf.setCastset( SituationHelper.getSituationCast( kcontext.getActivation(), type) );
    khelper.insertLogical(asf);

}

public static SituationType activateSituation(KnowledgeHelper khelper,
                                              SituationCastSet castset,
                                              Class<?> type) {

    long evn_timestamp =
khelper.getKnowledgeRuntime().getSessionClock().getCurrentTime();
    ActivateSituationEvent ase = new ActivateSituationEvent(evn_timestamp);

    SituationType sit = null;

    try {

        sit = (SituationType) type.newInstance();
        SetFieldsFromMatchedObjects(sit, castset);
        ase.setSituation(sit);
        sit.setActivation(ase);
        sit.setActive(true);
        khelper.getKnowledgeRuntime().insert(ase);
        khelper.getKnowledgeRuntime().insert(sit);

    } catch (InstantiationException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    } catch (IllegalAccessException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }

    return sit;
}

public static void deactivateSituation(KnowledgeHelper khelper, Object sit) {

    long evn_timestamp =
khelper.getKnowledgeRuntime().getSessionClock().getCurrentTime();
    DeactivateSituationEvent dse = new DeactivateSituationEvent(evn_timestamp);
    dse.setSituation((SituationType) sit);
    ((SituationType) sit).setDeactivation(dse);
    ((SituationType) sit).setActive(false);
    khelper.getKnowledgeRuntime().insert(dse);

khelper.getKnowledgeRuntime().update(khelper.getKnowledgeRuntime().getFactHandle(sit),
                                    sit);

}
}

```

