

Rafael Simonassi Amorim

Uma Abordagem Baseada em
Modelos para Gerenciamento de
Situações em Sistemas de
Processamento de Eventos
Complexos

Vitória - ES

18 de dezembro de 2015

Rafael Simonassi Amorim

Uma Abordagem Baseada em Modelos para Gerenciamento de Situações em Sistemas de Processamento de Eventos Complexos

Dissertação apresentada para
obtenção do grau de Mestre em Informática
pela Universidade Federal do Espírito Santo.

Orientadora:

Prof^a. Dr^a. Patrícia Dockhorn Costa

Coorientadora:

Prof^a. Dr^a. Roberta Lima Gomes

UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA

Vitória - ES

18 de dezembro de 2015

Dissertação de Mestrado sob o título “Uma Abordagem Baseada em Modelos para Gerenciamento de Situações em Sistemas de Processamento de Eventos Complexos”, defendida por Rafael Simonassi Amorim e aprovada em 18 de dezembro de 2015, em Vitória, Estado do Espírito Santo, pela banca examinadora constituída por:

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

Prof^a. Dr^a. Roberta Lima Gomes
Coorientadora

Prof. Dr. Cristiano André da Costa
Examinador externo

Prof. Dr. José Gonçalves Pereira Filho
Examinador interno

DEDICATÓRIA

Dedico este trabalho à minha família.

AGRADECIMENTOS

Gostaria de agradecer à minha orientadora Patrícia Dockhorn Costa por todo o empenho, dedicação e, acima de tudo, paciência durante todo o processo deste trabalho, desde a escolha do tema até a conclusão. Agradeço por toda disponibilidade e ajuda à minha coorientadora Roberta Lima Gomes. Ao professor João Paulo de Almeida, agradeço pela contribuição durante a elaboração dos modelos desenvolvidos neste trabalho. Agradeço à minha família como um todo, que me apoiou e me deu suporte do primeiro ao último dia. No mais, agradeço aos demais que, de alguma forma, contribuíram para que esse trabalho fosse concluído.

RESUMO

Aplicações que fazem uso de informações contextuais são ditas *aplicações sensíveis ao contexto*, reagindo e adaptando-se de acordo com as necessidades do usuário ou do sistema computacional. Um estado de interesse da aplicação sensível ao contexto é denominado *situação*. Para usufruir dos benefícios provenientes da utilização de situações em aplicações sensíveis ao contexto, é necessário prover suporte adequado tanto em tempo de projeto quanto em tempo de execução. O presente trabalho propõe uma metodologia para auxiliar a especificação, detecção e processamento de situações em sistemas CEP. Em tempo de projeto, a linguagem gráfica SML (*Situation Modeling Language*) é utilizada para modelar as situações e, em tempo de execução, foi desenvolvida uma plataforma CEP responsável por detectar e gerenciar o ciclo de vida das situações, denominada SIMPLE (*Situation Mapping Layer for Esper*). Foram implementadas transformações de modelos SML para código executável em SIMPLE, permitindo a detecção e processamento das situações descritas em SML. Para exemplificar a metodologia proposta, foi desenvolvida uma aplicação voltada para o cenário de detecção de fraudes bancárias. Por fim, foram realizados testes de desempenho com o intuito de mensurar o impacto da utilização da plataforma SIMPLE.

Palavras-chave: Situação, CEP, Complex Event Processing, ESPER.

ABSTRACT

Applications that make use of contextual information are said context-aware applications, reacting and adapting itself according to the needs of the user or the computer system. A state of interest of a context-aware application is called situation. To enjoy the benefits from the use of situations in context-aware applications, it is necessary to provide adequate support both in design time and in runtime. This paper proposes a methodology to support the specification, detection and processing of situations in CEP systems. At design time, the graphic language SML (Situation Modeling Language) is used to model situations and, at run time, a CEP platform was developed responsible for detecting and managing the life cycle of situations, called SIMPLE (Situation Mapping Layer for Esper). Transformations were implemented from SML models to executable code in SIMPLE, allowing the detection and processing of the situations described in SML. To illustrate the proposed methodology, an application dedicated to the detection of bank fraud scenario was developed. Finally, performance tests were conducted in order to measure the impact of using SIMPLE platform.

Sumário

1	INTRODUÇÃO	14
1.1	MOTIVAÇÃO	14
1.2	OBJETIVO	18
1.3	METODOLOGIA	19
1.4	VISÃO GERAL DA ABORDAGEM	23
1.5	ESTRUTURA DA DISSERTAÇÃO.....	24
2	CONCEITOS BÁSICOS E TRABALHOS RELACIONADOS	26
2.1	EVENTOS.....	26
2.2	COMPLEX EVENT PROCESSING	29
2.2.1	Arquitetura Geral.....	31
2.2.2	Event Processing Language (EPL)	36
2.3	SITUAÇÕES.....	38
2.4	SML	42
2.5	DESENVOLVIMENTO ORIENTADO A MODELOS.....	47
2.6	TRABALHOS RELACIONADOS	48
2.6.1	Situation Specification and Realization in Rule-Based Context-aware Applications	48
2.6.2	DS-EPL: Domain-Specific Event Processing Language	50
2.6.3	Learning from the past: automated rule generation for complex event processing	51
2.6.4	Situation-Aware Energy Control by Combining Simple Sensors and Complex Event Processing	52
2.6.5	Análise Comparativa	53
2.7	CONSIDERAÇÕES DO CAPÍTULO	55
3	ESPER.....	56
3.1	VISÃO GERAL DA PLATAFORMA	57
3.2	ESPER EPL.....	57
3.3	MODELO DE PROCESSAMENTO DE EVENTOS.....	59
3.4	EVENT PATTERNS	65

3.4.1	Operador Followed By.....	66
3.4.2	Operador Every	67
3.4.3	Combinando os operadores every e followed by.....	68
3.5	EXEMPLO DE APLICAÇÃO UTILIZANDO A PLATAFORMA ESPEER.....	70
3.6	CONSIDERAÇÕES DO CAPÍTULO	73
4	SIMPLE: SITUATION MAPPING LAYER FOR ESPEER	75
4.1	MODELO DE DETECÇÃO DE SITUAÇÕES EM CEP	75
4.2	SITUAÇÕES EM SIMPLE.....	79
4.3	INTEGRIDADE E COMPOSICIONALIDADE DE SITUAÇÕES.....	83
4.4	VISÃO GERAL E INSTRUÇÕES DE USO DA PLATAFORMA.....	85
4.4.1	Primeira Etapa.....	85
4.4.2	Segunda Etapa.....	86
4.5	IMPLEMENTAÇÃO DA PLATAFORMA (VISÃO INTERNA)	89
4.5.1	Situation.....	91
4.5.2	SituationDefinition	91
4.5.3	SituationManager	92
4.5.4	SituationListener e SituationManagerListener.....	92
4.6	CRIAÇÃO DE UM EVENTO DE ATIVAÇÃO DE SITUAÇÃO UTILIZANDO A PLATAFORMA SIMPLE	93
4.7	CONSIDERAÇÕES DO CAPÍTULO	96
5	REGRAS DE TRANSFORMAÇÃO.....	97
5.1	PADRÃO ENTIDADE-CONTEXTO INTRÍNSECO	98
5.2	PADRÃO ENTIDADE-CONTEXTO RELACIONAL	101
5.3	PADRÃO EXISTS	104
5.4	PADRÃO SITUAÇÃO COMPOSTA	107
5.5	CONSIDERAÇÕES DO CAPÍTULO	112
6	ESTUDO DE CASOS.....	113
6.1	CENÁRIO DE APLICAÇÃO.....	113
6.2	SITUAÇÕES DESCRITAS PARA O CENÁRIO PROPOSTO.....	115
6.3	APLICAÇÃO EXEMPLO.....	126
6.4	SUORTE FERRAMENTAL.....	129
6.4.1	Obeo Designer	129

6.4.2	Acceleo	131
6.5	CONSIDERAÇÕES DO CAPÍTULO	132
7	AVALIAÇÃO DE DESEMPENHO	133
7.1	METODOLOGIA	133
7.1.1	Configuração.....	134
7.1.2	Experimentos	135
7.2	RESULTADOS.....	140
7.2.1	Primeira bateria	142
7.2.2	Segunda bateria	144
7.2.3	Terceira bateria	145
7.2.4	Quarta bateria.....	147
7.2.5	Quinta bateria	148
7.3	CONSIDERAÇÕES DO CAPÍTULO	150
8	CONSIDERAÇÕES FINAIS.....	152
8.1	CONCLUSÕES.....	152
8.2	TRABALHOS FUTUROS	155
	REFERÊNCIAS.....	157
	ANEXO A – COMPONENTES INTERNOS DA PLATAFORMA SIMPLE.....	162
	ANEXO B – CÓDIGO UTILIZADO NA TRANSFORMAÇÃO DE MODELOS SML EM CÓDIGO SIMPLE.....	171

Lista de Figuras

Figura 1. Visão Geral da Abordagem	23
Figura 2. Eventos complexos	27
Figura 3. Relações entre intervalos temporais de Allen apresentados em (Schockaert, Cock, & Kerre, 2008).	29
Figura 4. <i>Logical view</i> da arquitetura de referência proposta pela EPTS (Paschke, Vincent, Alves, & Moxey, 2012).	33
Figura 5. <i>Functional view</i> da arquitetura de referência proposta pela EPTS (Paschke, Vincent, Alves, & Moxey, 2012).	35
Figura 6. Operadores temporais entre situações apresentados em (Pereira et al., 2013).	41
Figura 7. Exemplo do ciclo de vida de um tipo de situação (Pereira, 2013)	42
Figura 8. Exemplo de modelo de contexto	44
Figura 9. Exemplo de diagrama SML referente ao tipo de situação <i>Fever</i>	44
Figura 10. Exemplo de diagrama SML referente ao tipo de situação <i>LivingIn</i>	45
Figura 11. Exemplo de diagrama SML referente ao tipo de situação ILI	46
Figura 12. Exemplo de ocorrência no tempo de uma instância do tipo de situação ILI	47
Figura 13. Tabela comparativa dos trabalhos relacionados	53
Figura 14. Exemplo de uma <i>query</i> EPL simples	58
Figura 15. Exemplo de <i>join</i> entre dois fluxos diferentes de eventos	59
Figura 16. Exemplo de <i>query</i> utilizando janela	60
Figura 17. Modelo de processamento utilizando janela	61
Figura 18. Exemplo de <i>query</i> utilizando janela temporal	62
Figura 19. Modelo de processamento utilizando janela temporal	62
Figura 20. Exemplo de <i>query</i> utilizando filtros	63
Figura 21. Modelo de processamento utilizando filtros	64
Figura 22. Exemplo de <i>query</i> utilizando a cláusula <i>where</i>	64
Figura 23. Modelo de processamento utilizando a cláusula <i>where</i>	65
Figura 24. Exemplo do uso do operador <i>followed by</i>	67
Figura 25. Exemplo do uso do operador <i>every</i>	68
Figura 26. Exemplo do uso do operador <i>every</i> combinado com o operador <i>followed by</i>	68
Figura 27. Classe Java correspondente ao tipo de evento <i>OrderEvent</i>	71
Figura 28. Exemplo de <i>query</i> utilizando o tipo de evento <i>OrderEvent</i>	71
Figura 29. Exemplo de <i>Listener</i>	72
Figura 30. Exemplo de aplicação utilizando a plataforma Esper	73
Figura 31. Modelo de Detecção de Situações em CEP	76
Figura 32. Componente <i>Situation</i>	79
Figura 33. Primeira parte da implementação da classe <i>Fever</i>	81
Figura 34. Segunda parte da implementação da classe <i>Fever</i>	82
Figura 35. Regra de ativação referente ao tipo de situação <i>SituationC</i>	83

Figura 36. Exemplo de fluxo de dados de eventos do tipo <i>SituationA</i> e <i>SituationB</i>	84
Figura 37. Visão geral da plataforma SIMPLE	85
Figura 38. Implementação da classe Paciente no modelo de contexto da Figura 8	86
Figura 39. Diagrama de sequências dos passos necessários para utilizar a plataforma SIMPLE	87
Figura 40. Aplicação utilizando a plataforma SIMPLE para monitorar o tipo de situação <i>Fever</i>	87
Figura 41. Visão Interna da Plataforma SIMPLE	90
Figura 42. Diagrama de sequências referente a criação de um <i>Situation Creation Event</i>	95
Figura 43. Regra de transformação de ativação do padrão Entidade Contexto-Intrínseco	98
Figura 44. Regra de transformação de desativação do padrão Entidade-Contexto Intrínseco	100
Figura 45. Regra de transformação de ativação do padrão Entidade-Contexto Relacional	102
Figura 46. Regra de transformação de desativação do padrão Entidade-Contexto Relacional	103
Figura 47. Regra de transformação de ativação do padrão <i>Exists</i>	105
Figura 48. Regra de transformação de desativação do padrão <i>Exists</i>	107
Figura 49. Regra de transformação de ativação do padrão Situação Composta	109
Figura 50. Regra de transformação de desativação do padrão Situação Composta	111
Figura 51. Modelo de contexto para o cenário bancário (Mielke, 2013)	115
Figura 52. Situação <i>LoggedIn</i>	116
Figura 53. Primeira parte da classe Java correspondente ao tipo de situação <i>LoggedIn</i>	117
Figura 54. Segunda parte da classe Java correspondente ao tipo de situação <i>LoggedIn</i>	118
Figura 55. Situação <i>OngoingSuspiciousWithdrawal</i>	118
Figura 56. Regras de ativação e desativação geradas automaticamente para o tipo de situação <i>OngoingSuspiciousWithdrawal</i>	119
Figura 57. Situação <i>SuspiciousParallelLogin</i>	120
Figura 58. Exemplo de ocorrência no tempo para a situação <i>SuspiciousParallelLogin</i>	120
Figura 59. Primeira parte da classe Java correspondente ao tipo de situação <i>SuspiciousParallelLogin</i>	121
Figura 60. Segunda parte da classe Java correspondente ao tipo de situação <i>SuspiciousParallelLogin</i>	122
Figura 61. Situação <i>SuspiciousFarawayLogin</i>	123
Figura 62. Exemplo de ocorrência no tempo para a situação <i>SuspiciousFarawayLogin</i>	123
Figura 63. Regras de ativação e desativação geradas automaticamente para o tipo de situação <i>SuspiciousFarawayLogin</i>	124
Figura 64. Situação <i>AccountUnderObservation</i>	125
Figura 65. Exemplo de ocorrência no tempo para a situação <i>AccountUnderObservation</i>	125
Figura 66. Regras de ativação e desativação geradas automaticamente para o tipo de situação <i>AccountUnderObservation</i>	126
Figura 67. Aplicação simulando um cliente	127
Figura 68. Aplicação simulando uma central de monitoramento bancário	128

Figura 69. Configuração do computador utilizado ao longo dos experimentos	134
Figura 70. Configurações utilizadas na JVM ao longo dos experimentos	134
Figura 71. Exemplo de saída utilizando o <i>Esper Benchmark Kit</i>	136
Figura 72. Tabela contendo o resumo das baterias de testes realizadas	137
Figura 73. Regras de ativação e desativação dos tipos de situação <i>Sit1</i> e <i>Sit2</i>	138
Figura 74. Cenário onde X eventos do tipo <i>Market</i> são enviados ao servidor	139
Figura 75. Número de eventos esperados utilizando apenas a plataforma Esper	139
Figura 76. Número de eventos esperados utilizando as plataformas Esper e SIMPLE	140
Figura 77. Novas definições para as regras de desativação dos tipos de situação <i>Sit1</i> e <i>Sit2</i>	142
Figura 78. Comparativo entre os resultados da primeira bateria de testes.....	143
Figura 79. Comparativo entre os resultados da segunda bateria de testes	144
Figura 80. Comparativo entre os resultados da terceira bateria de testes	146
Figura 81. Comparativo entre os resultados da quarta bateria de testes	148
Figura 82. Comparativo entre os resultados da quinta bateria de testes.....	149
Figura 83. Componente <i>Situation</i>	163
Figura 84. Componente <i>SituationDefinition</i>	165
Figura 85. Componente <i>SituationManager</i>	166
Figura 86. Componente <i>SituationListener</i>	167
Figura 87. Componente <i>SituationManagerListener</i>	168

1 INTRODUÇÃO

1.1 MOTIVAÇÃO

Cada vez mais a computação torna-se presente na vida das pessoas. Atualmente é possível encontrar microprocessadores embutidos em objetos, computadores pessoais, dispositivos móveis, além de diversas outras tecnologias. Com o surgimento e expansão da internet, todos os dispositivos encontram-se, de alguma forma, conectados. A sensação de que a computação está em todo lugar e disponível a todo o momento deu origem ao conceito de computação ubíqua (ou pervasiva) (Hansmann, et al., 2003). A computação ubíqua permite que diferentes dispositivos e sistemas atuem de forma integrada e transparente ao usuário. Juntamente ao avanço em tecnologias de sensoriamento e o desenvolvimento de técnicas de extração de dados, a computação ubíqua viabiliza o uso de informações detalhadas, em tempo real, a respeito do usuário e seu respectivo *contexto* por parte das aplicações (Mcheick, 2004). “Contexto é qualquer informação que possa ser usada para caracterizar a situação de uma entidade. Uma entidade é uma pessoa, lugar ou objeto considerado relevante para a interação entre usuário e uma aplicação, incluindo o próprio usuário e a própria aplicação” (Abowd & Dey, 1999). Aplicações que fazem uso de informações contextuais são ditas *aplicações sensíveis ao contexto* (Costa, 2007). Essas aplicações reagem e se adaptam em tempo real, de acordo com as necessidades do usuário ou sistema computacional. Geralmente, essas aplicações estão interessadas não apenas nos valores associados ao contexto, mas no significado que o valor representa. Por exemplo, uma aplicação pode desejar ser notificada quando uma determinada pessoa encontra-se em estado febril, sem estar interessada no valor exato da temperatura que provocou tal febre. Esse “significado” representa um estado de interesse da aplicação sensível a contexto, e é denominado *situação*. (Costa, 2007).

Situações são entidades compostas cujos constituintes são outras entidades, suas propriedades e as relações em que estão envolvidos. Exemplos de situações são: “João está com febre”, “João teve uma febre durante os últimos seis meses”,

etc. Situações podem ser definidas em termos de situação mais simples, por exemplo a situação “febre intermitente de João”, especificada a partir da situação “João está com febre”, definida previamente.

Além disso, uma situação está associada a um intervalo de tempo, que determina sua duração. A existência de um intervalo de tempo garante um *ciclo de vida* associado à situação, na qual uma situação possui um indicador temporal de início (indicando o momento em que a situação começou a existir) e, possivelmente, um indicador temporal de fim (indicando o momento em que a situação deixou de existir). A presença de um ciclo de vida possibilita relacionar temporalmente diferentes ocorrências de situações.

O conceito de situação permite aos usuários (desenvolvedores, projetista, etc.) criar abstrações a respeito das entidades que compõem um determinado padrão de interesse. Ao invés de manipular entidades de baixo nível de abstração (ex. dados sensoriais que representam a temperatura de uma pessoa), é possível concentrar-se em padrões de alto nível de abstração construídos a partir de entidades de nível de abstração inferior (ex. a situação “febre de uma pessoa”).

Os trabalhos apresentados em (Costa 2007), (Costa 2012), (Pereira, 2013), (Rizzi, 2014) e (Sobral, 2015) visam usufruir dos benefícios provenientes da utilização de situação em aplicações sensíveis ao contexto. Esses trabalhos são complementares e avançam em uma mesma direção com o intuito de prover um *framework* completo para apoiar o desenvolvimento de aplicações sensíveis a situações, desde a etapa de modelagem até a etapa de implementação.

Segundo os trabalhos citados acima, para usufruir dos benefícios provenientes da utilização de situações em aplicações sensíveis ao contexto, é necessário prover suporte adequado (i) em tempo de projeto, (Costa 2007), (Costa, 2012), (Sobral, 2015); e (ii) em tempo de execução (Pereira, 2013), (Rizzi, 2014). Em tempo de projeto é necessário prover suporte para a especificação de situações num alto nível de abstração. Isso permite que desenvolvedores concentrem-se em padrões de alto nível de abstração, abstraindo escolhas tecnológicas ou entidades de baixo nível de abstração (ex: dados sensoriais). Além disso, em tempo de projeto, é necessário suporte para a definição de situações complexas (*i.e.*, compostas) em termos de

situações definidas previamente. Em tempo de execução é necessário prover suporte para detectar, gerenciar o ciclo de vida e processar adequadamente as situações especificadas durante a etapa de *design*, permitindo ao sistema reagir à ocorrência de cada situação.

Aplicações sensíveis ao contexto geralmente consistem em aplicações distribuídas, uma vez que possuem o objetivo de adquirir e prover informações contextuais a partir de diversas fontes distintas (Dey, Abowd, & Salber, 2001). Essa classe de aplicações deve lidar com fontes heterogêneas e desacopladas (em tempo, espaço e/ou sincronia) de informações contextuais, combinando as informações adquiridas a fim de prover um serviço transparente ao usuário. Aplicações sensíveis ao contexto devem lidar com desafios como: (i) gerenciar o recebimento e envio de dados de uma grande quantidade de fontes de dados; (ii) processar uma enorme quantidade de dados heterogêneos (pois existe uma grande quantidade de fonte de dados); (iii) apresentar tempo de resposta real sobre os dados processados, uma vez que as aplicações estão interessadas em reagir às informações contextuais no momento que as mudanças de estado são percebidas e (iv) apresentar um sistema sempre (ou maior parte do tempo) disponível.

Paralelamente, sistemas de CEP (*Complex Event Processing*) emergiram como uma área importante na indústria, sendo utilizados em diversos campos como simulação de eventos, bancos de dados ativos, gerenciamento de redes e raciocínio temporal (Eckert & Bry, 2009). Esses sistemas permitem o processamento contínuo de fluxos de eventos, lidando com grandes volumes de eventos em tempo real (Luckham D. , 2002). Em sistemas de CEP, eventos podem ser entendidos como “algo que aconteceu ou está contemplado para acontecer no domínio” (Etzion & Niblett, 2010). Sistemas de CEP interagem com uma grande quantidade de fontes de dados (*i.e.*, eventos) e consumidores, sendo estes heterogêneos e distribuídos, os quais têm como objetivo observar o mundo externo e reagir a ele (Cugola & Margara, 2012). Diversos sistemas de CEP surgiram ao longo dos últimos anos, por exemplo, Esper (Esper, 2006), PADRES (Fidler, Jacobsen, Li, & Mankovskii, 2005), Tibco (Tibco, 2015), Coral8 (Coral8, 2009), dentre outras.

Sistemas de CEP provêm mecanismos para a criação de eventos complexos (*i.e.*, compostos) a partir da composição e/ou agregação de outros eventos. Eventos são relacionados a partir da especificação de padrões, nos quais representam estados particulares de interesse sobre os quais o sistema deseja agir (ou reagir). Em sistemas CEP, padrões particulares de interesse são definidos a partir da utilização de uma EPL (*Event Processing Language*) (Etzion & Niblett, 2010). Tipicamente, uma EPL possui construções que permitem a especificação de um padrão de interesse por meio de filtros e operadores de agregação, além da possibilidade de relacionar diferentes fluxos de eventos. Também é possível especificar padrões considerando aspectos de temporalidade e causalidade.

Em geral, EPLs utilizadas em sistemas de CEP não possuem construções específicas de domínio, consistindo em linguagens de propósitos gerais (Bruns, Dunkel, Lier, & Masbruch, 2014). A utilização de construções de propósitos gerais em EPL, a fim de permitir independência entre domínios, leva a construção de padrões mais extensos e complexos, trazendo desvantagens como: (i) perda de expressividade; (ii) falta de escritabilidade e (iii) falta de legibilidade, dentre outros. Desenvolvedores devem passar por uma curva de aprendizado a fim de explorar todas as funcionalidades disponíveis na EPL correspondente. Por vezes, especialistas do domínio, ou seja, aqueles que possuem conhecimento a respeito dos padrões de interesse, não possuem o conhecimento técnico desejado para representar os padrões de interesse, por meio de uma EPL, no sistema CEP utilizado.

Outro ponto importante a ser destacado consiste na falta de suporte adequado para o gerenciamento e detecção de situações em sistemas CEP. Sistemas CEP são desenvolvidos visando à manipulação de eventos, que representam estados de interesse pontuais (JBoss, 2014), não possuindo um ciclo de vida associado, como ocorre em sistemas sensíveis a situações. Além disso, geralmente, sistemas CEP pecam quanto à composicionalidade, não oferecendo suporte adequado para a definição de padrões a partir de padrões previamente definidos. Como discutido anteriormente, sistemas sensíveis a situações devem permitir a composição de situações definidas previamente.

As características de sistemas CEP apresentadas, nos quais permitem a manipulação de uma enorme quantidade de dados (*i.e.*, eventos) heterogêneos com tempo de resposta real vão ao encontro dos requisitos necessários para o desenvolvimento de aplicações sensíveis ao contexto e, conseqüentemente, sensíveis a situações. Além disso, os mecanismos para a especificação de padrões de interesse disponibilizados em sistemas de CEP podem ser adaptados de forma a permitir a representação de ocorrências de situações a partir de dados contextuais.

Uma vez que sistemas de CEP não provêm suporte adequado para o gerenciamento de situações e apresentam requisitos nos quais são compatíveis aos desafios encontrados na realização de aplicações sensíveis ao contexto, o presente trabalho tem como objetivo propor uma metodologia para auxiliar a especificação, detecção e processamento de situações em sistemas CEP, tanto em tempo de projeto quanto em tempo de execução. Tal metodologia deve considerar os desafios discutidos anteriormente: (i) dificuldade em utilizar sistemas de CEP para representar os padrões de interesse, por meio de EPLs; (ii) falta de suporte a especificação de situações, levando em consideração aspectos como composicionalidade; (iii) falta de suporte à detecção e gerenciamento de situações;

Durante o desenvolvimento dessa metodologia, verificou-se que as especificações EPL para definir situações (simples e compostas) eram muito complexas e, por vezes, inviáveis de serem definidas manualmente. A fim de resolver este problema, a metodologia também propõe a utilização de técnicas de desenvolvimento orientado a modelos (Almeida, 2006), para permitir (i) a especificação de situações em um alto nível de abstração, e (ii) o mapeamento automático destas especificações para código em uma EPL.

1.2 OBJETIVO

O presente trabalho tem como objetivo geral propor uma metodologia para auxiliar a especificação, detecção e processamento de situações em sistemas de CEP. Esse objetivo pode ser subdividido nos seguintes objetivos específicos:

- 1) Definir conceitos que sirvam como fundamentação para a realização de situações no domínio de CEP. Uma vez que sistemas de CEP manipulam apenas eventos, tais conceitos devem explicitar as relações necessárias para a definição de situações a partir da utilização de eventos. Além disso, os conceitos propostos devem ser adequados para permitir a especificação de situações considerando aspectos temporais e composicionais.
- 2) Desenvolver uma plataforma CEP que permita a especificação de situações (i) em tempo de projeto; e (ii) em tempo de execução. Em tempo de projeto deseja-se permitir que desenvolvedores modelem situações em um alto nível de abstração, de forma totalmente independente das escolhas tecnológicas. Em tempo de execução, deseja-se permitir a detecção e gerenciamento das situações modeladas. Tal plataforma deve considerar aspectos temporais das situações e composicionalidade (*i.e.*, a definição de situações a partir de situações definidas previamente). O desenvolvimento de tal plataforma deve considerar como base a utilização de plataformas CEP existentes.
- 3) Transformar os modelos contendo as definições das situações do domínio, descritos durante a etapa de *design*, em código específico da plataforma de situações desenvolvida (objetivo 2), a fim de facilitar e automatizar o processo de especificação de situações.
- 4) Analisar o desempenho da plataforma de situações desenvolvida no objetivo 2, a fim de mensurar o impacto introduzido por essa plataforma sobre a plataforma CEP utilizada como base.

1.3 METODOLOGIA

A etapa inicial para o desenvolvimento deste trabalho, guiada pelo objetivo 1, consistiu no estudo de propostas de plataformas CEP e do uso de situações em aplicações sensíveis ao contexto relatadas na literatura, a fim de obter entendimento

a respeito dos conceitos e requisitos necessários para o desenvolvimento de aplicações nessas áreas.

Nessa etapa, o trabalho apresentado em (Costa, 2007) serviu de inspiração para o desenvolvimento da abordagem proposta. (Costa, 2007) propõe um *framework* para desenvolvimento de aplicações sensíveis a contexto, tendo como ponto de partida a definição de modelos de contexto e situação, especificados por meio de diagramas de classe UML e restrições OCL.

Com relação a conceitos básicos de sistemas CEP, o trabalho apresentado em (Etzion & Niblett, 2010), bem como os documentos gerados pela *Event Processing Technical Society* (EPTS) (Event Processing Glossary, 2011) foram amplamente utilizados. A EPTS consiste na iniciativa de um grupo de organizações e indivíduos que se reuniram com o objetivo comum de propor uma padronização na área de processamento de eventos e, conseqüentemente, na área de CEP. Atualmente, todas as páginas Web referentes à EPTS encontram-se indisponíveis, evidenciando o fracasso da iniciativa. O trabalho apresentado em (Etzion & Niblett, 2010), um dos participantes de EPTS, também serviu como fonte de entendimento dos conceitos relacionados à área de CEP.

Os estudos realizados resultaram num modelo de detecção de situações em CEP (seção 4.1). Esse modelo levou a um melhor entendimento a respeito das áreas de CEP e de aplicações sensíveis ao contexto. Além disso, foi possível compreender as relações necessárias para a definição de situações a partir da utilização de eventos. Os estudos realizados, juntamente com o modelo desenvolvido, levaram também ao levantamento dos requisitos necessários para o desenvolvimento da plataforma proposta no objetivo 2.

Concluída a primeira etapa no qual obteve-se entendimento a respeito dos conceitos e requisitos necessários para o desenvolvimento de aplicações nessas áreas, iniciou-se a etapa de desenvolvimento da plataforma de situações em CEP, baseado no objetivo 2. Essa plataforma (denominada SIMPLE - *Situation Mapping Layer for Esper*) foi projetada a partir da utilização de uma plataforma CEP já existente, a plataforma Esper (Esper, 2006).

Diversos motivos levaram à escolha da plataforma Esper como base para o desenvolvimento da plataforma SIMPLE. Esper consiste numa plataforma de CEP *open source*, a qual possui uma ampla comunidade e suporte por parte de sua empresa mantenedora, *EsperTech*. Sua documentação é bem detalhada e organizada, descrevendo informações necessárias para seu entendimento e utilização, bem como métricas para avaliação de desempenho utilizando tal plataforma. Esper é distribuída na linguagem Java, uma linguagem amplamente utilizada com a qual grande parte dos desenvolvedores encontra-se familiarizada. Além disso, tal plataforma é disponibilizada na forma de uma biblioteca, podendo ser facilmente configurada e incorporada ao ambiente de trabalho.

Esper possui uma EPL altamente expressiva, denominada Esper EPL, que possui uma sintaxe baseada na linguagem SQL (também amplamente utilizada), facilitando seu uso. Essa linguagem permite a definição de padrões complexos, por meio de construções que relacionam diferentes fluxos de eventos, por exemplo, agregação e composição, além da possibilidade de relacionar tais fluxos com base no aspecto temporal.

Inicialmente, optou-se por utilizar a plataforma PADRES (Fidler, Jacobsen, Li, & Mankovskii, 2005) como base para o desenvolvimento da plataforma de situações em CEP. Apesar de apresentar-se como uma plataforma robusta e abordada em diversos trabalhos (Domínguez, Lloret, Pérez, Rodríguez, Rubio, & Zapata, 2007) (Cugola & Margara, 2009) (Cugola & Margara, 2012), foram encontradas dificuldades ao utilizar tal plataforma, uma vez que sua documentação é escassa e limitada, apresentando apenas detalhes superficiais de uso. Alguns componentes não apresentaram o comportamento desejado, e a falta de documentação robusta e detalhada ou de uma comunidade ativa, impossibilitaram sanear dúvidas recorrentes. A EPL utilizada pela plataforma PADRES deixa a desejar quanto à expressividade, limitando-se à especificação de padrões apenas por meio de tuplas na forma (atributo, relação, valor), além de não permitir relacionar diferentes fluxos temporalmente. Após meses de trabalho na tentativa de utilizar a plataforma PADRES, por fim, sua substituição pela plataforma Esper mostrou-se uma escolha acertada.

A fim de usufruir dos benefícios provenientes da utilização de situações, em tempo de projeto, baseado no objetivo 2, torna-se necessário prover tal suporte no nível de modelagem, permitindo que desenvolvedores modelem situações em um alto nível de abstração, de forma totalmente independente das escolhas tecnológicas. Para prover tal suporte, o presente trabalho utilizou SML (Situation Modeling Language) (Mielke, 2013), que consiste numa linguagem gráfica para a modelagem de situações, apresentando construções que permitem representar situações simples e complexas, num alto nível de abstração. As situações modeladas em SML foram exemplificadas por meio de um cenário de detecção de fraudes bancárias, que também foi utilizado pelo trabalho apresentado por (Mielke, 2013).

Conforme discutido anteriormente, durante o desenvolvimento dessa metodologia, verificou-se que as especificações EPL para definir situações, sejam elas simples ou compostas, apresentavam-se complexas e, por vezes, inviáveis de serem definidas manualmente, principalmente para usuário que não tem conhecimento técnico sobre as tecnologias utilizadas. A fim de resolver esse problema, baseado no objetivo 3, a metodologia propõe a utilização de técnicas de desenvolvimento orientado a modelos (Almeida, 2006), para permitir (i) a especificação de situações em um alto nível de abstração, e (ii) o mapeamento automático destas especificações para código em uma EPL. A partir dos modelos de situações especificados em SML, foram definidas transformações dos modelos para código da plataforma SIMPLE. Além disso, foi implementada uma aplicação que faz uso do código gerado automaticamente para detectar situações, analogamente ao que foi feito em (Mielke, 2013), a fim de exemplificar a viabilidade da abordagem proposta.

Por fim, com base no objetivo 4, foram elaborados e executados testes sobre a plataforma SIMPLE, com o intuito de mensurar o impacto do mesmo sobre a plataforma Esper. Os testes realizados tomaram como base testes apresentados pelo *benchmark kit*¹ da plataforma Esper, e são discutidos no capítulo 7.

¹ <http://www.espertech.com/esper/performance.php>

1.4 VISÃO GERAL DA ABORDAGEM

A abordagem aqui proposta se apoia na geração automática de código a partir de modelos descritos em SML. O código gerado permite que a plataforma Esper (Esper, 2006) seja capaz de detectar e gerir o ciclo de vida das situações descritas nos modelos SML, por meio da utilização da plataforma SIMPLE (capítulo 4). A Figura 1 apresenta uma visão geral da abordagem utilizada, em que é possível observar a divisão da solução em três camadas distintas.

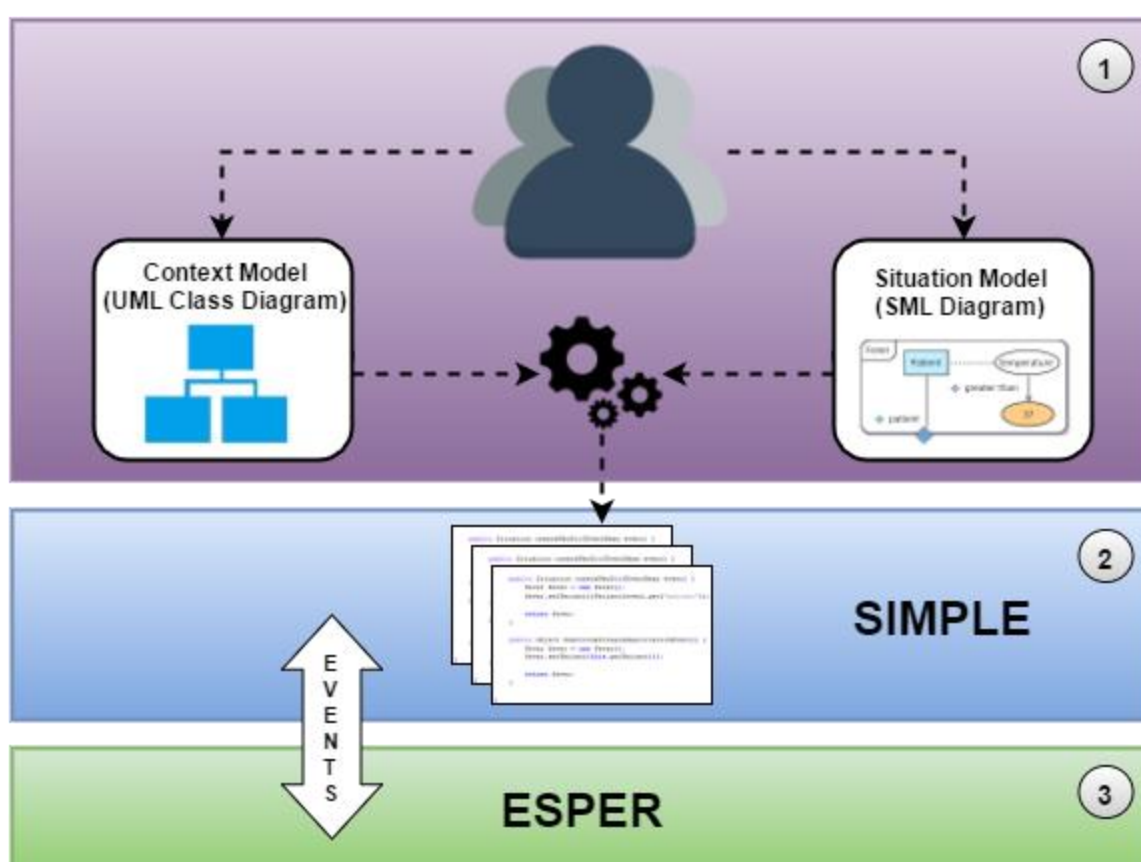


Figura 1. Visão Geral da Abordagem

Na camada superior (Figura 1-1), a camada de especificação, encontram-se os desenvolvedores, responsáveis por especificar os modelos de contexto (diagramas UML) e situação (diagramas SML), provendo suporte à especificação de situações em tempo de projeto. Sobre os modelos de contexto e de situações são aplicadas transformações responsáveis por gerar, automaticamente, código necessário para que a plataforma SIMPLE seja capaz de detectar e gerenciar de forma adequada as situações modeladas na camada de especificação. A plataforma SIMPLE representa

a segunda camada (Figura 1-2), a camada de realização, provendo suporte à detecção e processamento de situações por meio dos eventos recebidos e enviados à plataforma Esper (representando a última camada (Figura 1-3)).

1.5 ESTRUTURA DA DISSERTAÇÃO

Este trabalho está estruturado da seguinte forma:

O Capítulo 2 apresenta os conceitos básicos necessários para o entendimento deste trabalho. Além disso, são apresentados e discutidos os trabalhos relacionados.

O Capítulo 3 apresenta uma visão geral da plataforma Esper, utilizada como base para o desenvolvimento deste trabalho, discutindo aspectos referentes ao seu modelo de processamento de eventos e suas principais construções utilizadas para a definição dos padrões e regras de transformações apresentados no capítulo 5.

O Capítulo 4 apresenta a plataforma de situações em CEP, denominada SIMPLE. Esse capítulo apresenta o modelo de detecção de situações em CEP, utilizado como ponto de partida para o desenvolvimento de tal plataforma. Além disso, são apresentadas visões distintas da plataforma: (i) uma visão geral, voltada para os usuários finais da plataforma e (ii) uma visão mais técnica, voltada para desenvolvedores, que apresenta detalhes a respeito da implementação da plataforma.

O Capítulo 5 apresenta e detalha os padrões principais utilizados para definir a ocorrência de uma situação utilizando regras Esper EPL. Para cada padrão são apresentadas as regras de transformação utilizadas na geração de regras de ativação e desativação a partir de modelos SML.

O Capítulo 6 apresenta um estudo de casos, em que a abordagem proposta é ilustrada por meio de um cenário voltado para a detecção de fraudes bancárias. São apresentados e discutidos os tipos de situações definidos para tal cenário. Além disso, é apresentada uma aplicação, baseada no cenário proposto, desenvolvida a

partir do código gerado automaticamente segundo as regras de transformação apresentadas no capítulo 5.

O Capítulo 7 apresenta uma avaliação de desempenho da plataforma SIMPLE. Foi realizado o mesmo experimento duas vezes, a primeira vez utilizando o apenas a plataforma Esper, e a segunda vez utilizando a plataforma SIMPLE, com acréscimo de alguns tipos de situação a serem detectados. A fim de mensurar o impacto, em relação à latência e vazão (*throughput*), sobre a plataforma Esper ocasionado pela utilização da plataforma SIMPLE, ambos os resultados foram comparados, analisados e discutidos.

Por fim, no Capítulo 8 são apresentadas as conclusões e os trabalhos futuros.

2 CONCEITOS BÁSICOS E TRABALHOS RELACIONADOS

Este capítulo apresenta os principais conceitos nas seguintes áreas: aplicações sensíveis a contexto, sistemas baseados em eventos e desenvolvimento orientado a modelos (MDD). Esses conceitos são fundamentais para o entendimento do presente trabalho, bem como da abordagem proposta. Além disso, este capítulo discute trabalhos relacionados aos objetivos propostos no capítulo anterior, sendo que alguns deles serviram como inspiração para a metodologia proposta. São abordados diferentes trabalhos que lidam com a especificação e/ou realização de situações.

O capítulo encontra-se organizado da seguinte forma: a Seção 2.1 discute o conceito de evento adotado pelo presente trabalho; a Seção 2.2 apresenta conceitos relacionados a sistemas baseados em eventos, mais precisamente sistemas de processamento de eventos complexos (CEP); a Seção 2.3 discute o conceito de situações; a Seção 2.4 apresenta a linguagem SML, utilizada para modelar os tipos de situações; a Seção 2.5 apresenta conceitos da área de desenvolvimento orientado a modelos (MDD); a Seção 2.6 apresenta e discute os trabalhos relacionados; e, por fim, a Seção 2.7 apresenta as considerações do capítulo.

2.1 EVENTOS

Segundo (Etzion & Niblett, 2010), “um evento é uma ocorrência dentro de um sistema ou domínio em particular; é algo que aconteceu ou está contemplado como tendo acontecido nesse domínio”. Em um sistema de computação, um evento corresponde a uma entidade computacional que representa esta ocorrência.

Eventos podem ser classificados em eventos simples ou complexos (também chamados de eventos compostos ou eventos derivados) (Fülöp, et al., 2010). Eventos simples são eventos atômicos, ou seja, eventos primitivos que não podem ser decompostos em outros eventos. Eventos complexos são eventos criados a

partir da composição ou agregação de outros eventos, sejam estes simples ou até mesmo outros eventos complexos.

Em aplicações que fazem uso de eventos é comum observar uma estrutura e semântica comuns entre os diversos eventos. Por exemplo, eventos que monitoram a temperatura de um congelador. Todos os eventos de temperatura possuem o mesmo tipo de informação, diferenciando-se apenas em seu respectivo valor de temperatura associado. Dessa forma, ao invés de definir a estrutura individual de cada evento, é especificada a estrutura de toda essa classe de eventos, por exemplo, a classe “eventos de temperatura do congelador”.

A especificação da estrutura comum a todos os eventos de uma mesma classe é denominada tipo de evento (*event type*) (Etzion & Niblett, 2010). A representação computacional de um evento representa uma instância de um tipo de evento. Por exemplo, “pedido” é um exemplo de um tipo de evento, enquanto que eventos contendo informações sobre “pedido de um determinado produto” representam instâncias do tipo de evento “pedido”. Um evento é dito detectado a partir do momento em que a aplicação tem conhecimento de sua ocorrência, gerando uma instância do mesmo.

Um evento complexo pode ser visto como uma instância do padrão que o define. A Figura 2 ilustra a criação de um evento complexo. Eventos (*Event*), simples ou complexos, são usados como entrada (seta 1, Figura 2) para um padrão de eventos (*Event Pattern*). O padrão relaciona os diversos eventos, tendo como saída (seta 2, Figura 2) um evento complexo (*Complex Events*), representando a ocorrência de tal padrão.

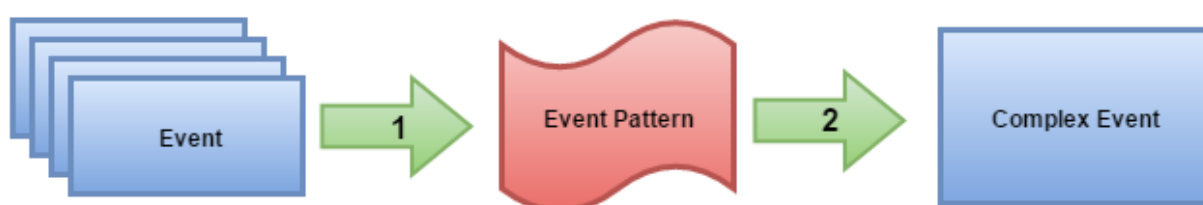


Figura 2. Eventos complexos

Tomemos como exemplo um sistema de computação de uma cafeteria. Possíveis eventos simples são representados por: “o café está pronto” ou “o

hambúrguer está aquecido”. A composição de ambos os eventos pode dar origem a um evento complexo representado por “o pedido está completo”. Alguns eventos são observados de forma direta, por exemplo, o evento “o estoque de café acabou”. Por outro lado, alguns eventos complexos são inferidos a partir da observação de outros eventos. Por exemplo, o evento “o consumo de café está aumentando”, observado a partir da ocorrência, acima da média, do número de eventos do tipo “o café está pronto”.

Eventos também podem ser classificados conforme sua dimensão temporal, como eventos pontuais ou eventos com duração (Etzion & Niblett, 2010). Eventos pontuais possuem apenas um *timestamp* associado, indicando sua ocorrência num único ponto do tempo, sem duração. Um *timestamp* pode representar: (i) o instante no tempo que representa a ocorrência do evento; (ii) o instante no tempo que o evento foi detectado pelo sistema ou; (iii) o instante no tempo que o evento foi processado pelo sistema. Eventos com duração estão associados a um intervalo de tempo definido por dois *timestamps*: um indicando seu início e outro indicando seu fim. A utilização de *timestamps* impõe uma ordenação temporal sobre os eventos, além de permitir a especificação de padrões que relacionem eventos passados (*i.e.*, onde seu *timestamp* ou intervalo representam valores menores que o tempo atual) e eventos presentes.

Normalmente, essas relações temporais são realizadas por meio das relações de álgebra de intervalos de Allen (Allen, 1983). Segundo (Allen, 1983), um intervalo consiste num par de pontos ordenados no tempo, no qual o primeiro ponto possui valor menor do que o segundo. Assumindo um intervalo T , o ponto menor é denominado como t_- , enquanto que o ponto maior é denominado como t_+ . Em (Allen, 1983) é definido um conjunto de 13 relações existentes entre dois intervalos: *before*, *after*, *during*, *equals*, *meets*, *met-by*, *overlaps*, *overlapped-by*, *finishes*, *finished-by*, *starts*, *startedby* e *coincides*. A Figura 3 ilustra as 13 relações temporais definidas pela álgebra de intervalos de Allen, baseado nos intervalos $A = [a_-, a_+]$ e $B = [b_-, b_+]$.

Name	Definition
before	$b(A, B) \equiv a^+ < b^-$
overlaps	$o(A, B) \equiv a^- < b^- \text{ and } b^- < a^+ \text{ and } a^+ < b^+$
during	$d(A, B) \equiv b^- < a^- \text{ and } a^+ < b^+$
meets	$m(A, B) \equiv a^+ = b^-$
starts	$s(A, B) \equiv a^- = b^- \text{ and } a^+ < b^+$
finishes	$f(A, B) \equiv a^+ = b^+ \text{ and } b^- < a^-$
equals	$e(A, B) \equiv a^- = b^- \text{ and } a^+ = b^+$
after	$bi(A, B) \equiv b(B, A)$
overlapped-by	$oi(A, B) \equiv o(B, A)$
contains	$di(A, B) \equiv d(B, A)$
met-by	$mi(A, B) \equiv m(B, A)$
started-by	$si(A, B) \equiv s(B, A)$
finished-by	$fi(A, B) \equiv f(B, A)$

Figura 3. Relações entre intervalos temporais de Allen apresentados em (Schockaert, Cock, & Kerre, 2008).

2.2 COMPLEX EVENT PROCESSING

Sistemas de CEP (*Complex Event Processing*) emergiram como uma área importante na indústria, sendo utilizados em diversos campos como simulação de eventos, bancos de dados ativos, gerenciamento de redes e raciocínio temporal (Eckert & Bry, 2009). O objetivo principal de sistemas de CEP consiste em identificar eventos significativos (por exemplo, comportamentos suspeitos ou que apresentam um padrão) e responder a eles o mais rápido possível (Wikipedia, 2015). Em sistemas de CEP, eventos podem ser entendidos como “algo que aconteceu ou está contemplado para acontecer no domínio” (Etzion & Niblett, 2010). Esses sistemas provêm mecanismos para especificação de tipos de eventos a partir da composição e/ou agregação de outros tipos de eventos. Em outras palavras, é possível dizer que sistemas de CEP geram conhecimento de nível mais alto a partir de eventos de nível inferior (Fülöp, et al., 2010). Diversos sistemas de CEP surgiram ao longo dos últimos anos, por exemplo, Esper (Esper, 2006), PADRES (Fidler, Jacobsen, Li, & Mankovskii, 2005), Tibco (Tibco, 2015), Coral8 (Coral8, 2009), dentre outras.

Eventos são recebidos e enviados pelo sistema de CEP por meio de diferentes fluxos contínuos de eventos, permitindo que grandes volumes de eventos sejam

avaliados no momento de sua ocorrência (*i.e.*, em tempo real) (Luckham D. , 2002). Fluxos de eventos podem representar conjuntos finitos ou infinito de eventos. Um conjunto finito é definido por meio do conceito de janelas. Uma janela representa um segmento delimitado de um fluxo de eventos, definindo um subconjunto (Etzion & Niblett, 2010). Um exemplo de janela é dada por “os últimos cinco eventos recebidos”, que permite que o processamento do sistema seja voltado apenas para os cinco últimos eventos recebidos, de forma que o sistema de CEP concentre-se nos dados relevantes, não sendo necessário processar todos os eventos recebidos ao longo do tempo. Janelas podem ser classificadas ainda como janelas temporais, como a janela definida como “os eventos nos últimos dez minutos”, representando o subconjunto formado por todos os eventos recebidos nos últimos 10 minutos, com base no *timestamp* associado a cada evento.

Para permitir a comunicação entre diversas fontes heterogêneas, com fluxo contínuo de dados, é comum que sistemas de CEP façam uso do paradigma *Publish/Subscribe* (Eugster, Felber, Guerraoui, & Kermarrec, 2003). É possível encontrar na literatura trabalhos (Feltrinelli, 2010) (Cugola & Margara, 2012) (Baptista et al. 2013) que consideram CEP como uma evolução natural do paradigma *Publish/Subscribe*.

Sistemas *Publish/Subscribe* tradicionais consideram cada evento separadamente e filtram os mesmos utilizando seu conteúdo (*content-based*) ou tópico (*topic-based*), ou tipo (*type-based*) para decidir quais eventos são relevantes para os *Subscribers*. Em sistemas CEP essa funcionalidade é estendida por meio de eventos complexos, permitindo que *subscribers* (*i.e.*, consumidores) manifestem seus interesses por meio de eventos compostos (*i.e.*, eventos complexos) (Cugola & Margara, 2012). Eventos complexos são eventos cuja ocorrência (*i.e.*, sua detecção) depende da ocorrência de outros eventos, por exemplo, o fato de um incêndio ser detectado a partir da medição de temperatura acima do esperado para três ou mais sensores diferentes.

A relação entre os diversos eventos que originam um evento complexo é realizada por meio da utilização de padrões, conforme ilustrado na Figura 2 e discutido na seção 2.1. Em sistemas CEP, os padrões particulares de interesse, utilizados para relacionar eventos e produzir eventos complexos, são definidos a

partir da utilização de EPLs (*Event Processing Language*) (Etzion & Niblett, 2010), nas quais permitem a detecção de padrões aplicando operações sobre os fluxos de eventos, por exemplo, agregação, composição, filtros, janelas, *join*, etc. EPLs são abordadas e discutidas em detalhes na seção 2.2.2.

Sistemas de CEP interagem com uma grande quantidade de fontes de dados e consumidores, sendo estes heterogêneos e distribuídos, nos quais têm como objetivo observar o mundo externo e reagir a ele (Cugola & Margara, 2012). A capacidade de lidar com uma enorme fonte de dados heterogêneos e distribuídos acarretam na necessidade de garantir requisitos, por exemplo: (i) escalabilidade, uma vez que sistemas de CEP lidam com uma enorme quantidade de fontes de dados; (ii) grande capacidade de processamento, uma vez que uma enorme quantidade de fonte de dados tende a produzir uma enorme quantidade de dados (*i.e.*, eventos simples e complexos); (iii) resposta em tempo real, pois normalmente aplicações que utilizam sistemas de CEP desejam reagir aos eventos do mundo externo assim que os mesmos são percebidos, ou seja, no momento de sua ocorrência (ou o mais rápido possível) e (iv) disponibilidade, garantindo que tanto o sistema de CEP quanto as aplicações que o usam estejam disponíveis a maior parte do tempo.

2.2.1 Arquitetura Geral

Atualmente, não existe uma padronização para a área de CEP. Cada fabricante desenvolve sua própria solução e, conseqüentemente, sua própria arquitetura (Cugola & Margara, 2012). Baseado nisso, a *Event Processing Technical Society* (EPTS) foi criada com o objetivo de propor uma padronização na área de processamento de eventos, conforme discutido anteriormente na seção 1.3. Os esforços realizados pela EPTS originaram, dentre outras coisas, uma arquitetura de referência voltada para a área de processamento de eventos, e conseqüentemente, para a área de CEP (Paschke, Vincent, Alves, & Moxey, 2012).

A arquitetura de referência proposta em (Paschke, Vincent, Alves, & Moxey, 2012) foca nos elementos comuns, e suas relações, necessários para permitir a

realização e implementação de sistemas de processamento de eventos, sem especificar quaisquer detalhes a respeito de escolhas tecnológicas. Seu objetivo consiste em apresentar os principais componentes e funções comumente encontrados na maioria das arquiteturas de sistemas de processamento de eventos. Essa arquitetura é dividida em duas *views*: a *logical view* e a *functional view*. Uma *view* consiste em “uma representação de todo o sistema a partir da perspectiva de um conjunto relacionado de preocupações” (Paschke, Vincent, Alves, & Moxey, 2012).

A *logical view* descreve a camada lógica de *Event Processing Agents* (EPAs) em uma *Event Processing Network* (EPN). Segundo (Luckham, et al., 2011) um EPA corresponde a “uma entidade que processa eventos”, podendo realizar computações e operações sobre os mesmos, como agregação, aplicação de filtros e detecção de padrões de eventos. Um EPA pode exercer tanto o papel de consumidor como de produtor de eventos. Por exemplo, um EPA pode exercer o papel de consumidor, ao receber eventos externos ao sistema, por exemplo, dados sensoriais. Uma vez que o EPA realize algum processamento sobre os dados recebidos, por exemplo, agregação, o mesmo pode exercer o papel de produtor, fornecendo os dados processados a outro módulo do sistema, ou até mesmo um outro EPA. Já um EPN consiste num conjunto de EPAs e os canais utilizados para sua comunicação (Luckham, et al., 2011). É interessante observar que as definições de EPA e EPN são recursivas: uma EPN pode representar um EPA do ponto de vista de uma EPN de maior nível de abstração.

A Figura 4 apresenta uma visão geral da *logical view*. Essa visão pode ser descrita como uma visão abstrata, que descreve a distribuição lógica de cada EPA, suas relações, bem como qualquer mecanismo de rede de tratamento de eventos ou de distribuição, a partir de uma perspectiva de interesse (de um sistema ou subsistema). Nessa figura é possível observar um produtor (*Emitter*) (por exemplo, um sensor), responsável por enviar dados (*Data*) ao sistema (nesse caso, a EPN). Eventos são distribuídos pela EPN por meio de nós (*Root Events*) até que, eventualmente, são recebidos por EPAs (*Event Processor*) através de um “barramento de eventos” (*Event Bus*). Após processar um evento, um EPA pode ter como saída um único evento ou um conjunto de eventos, gerados a partir do evento

de entrada. EPAs podem também utilizar fontes externas (*External*), por exemplo, bancos de dados, a fim de utilizar dados históricos na composição de um novo evento. Eventos gerados por um EPA são enviados novamente ao meio de comunicação, podendo ser processados por outros EPAs ou consumidores (*Consumer*). Uma vez recebido um evento ou conjunto de eventos de seu interesse, consumidores reagem de forma apropriada (*Reaction*).

A *lógica* *view* pode ser utilizada para o entendimento melhor a respeito do sistema como um todo (e do fluxo dos eventos emitidos pelo sistema), por exemplo, apresentando uma visão geral das fontes de entrada e saída do sistema, bem como um melhor entendimento a respeito do papel de cada EPA, como estão organizados e distribuídos.

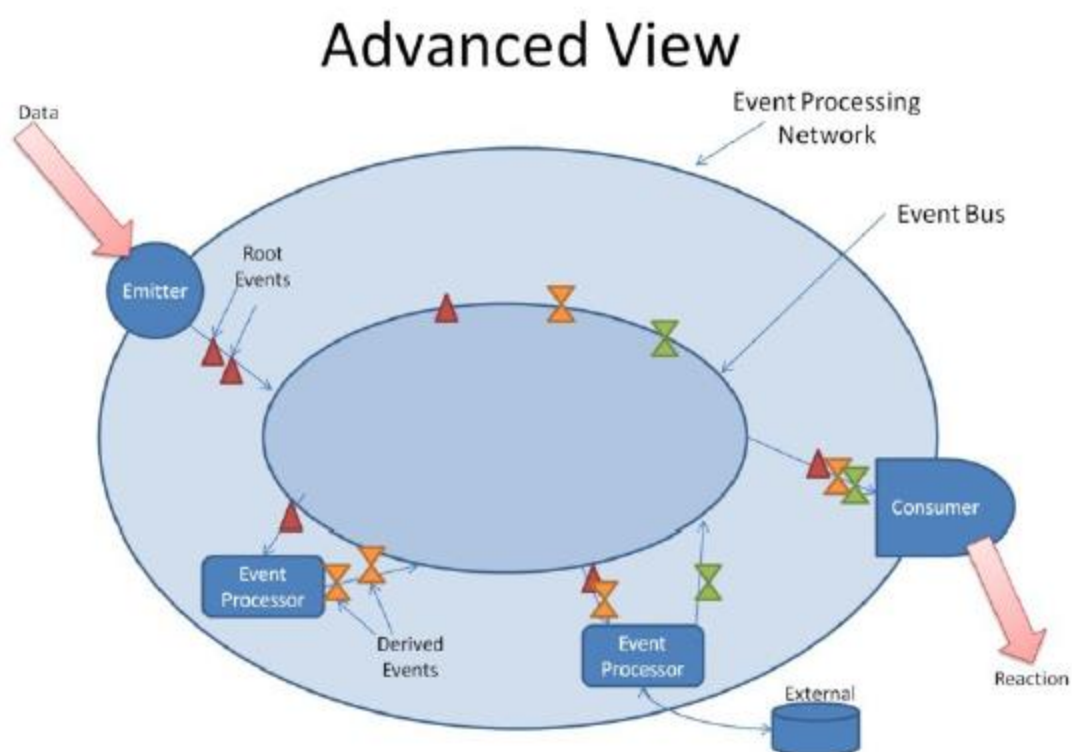


Figura 4. *Logical view* da arquitetura de referência proposta pela EPTS (Paschke, Vincent, Alves, & Moxey, 2012).

Por sua vez, a *functional view* tem como foco as funcionalidades que podem ser requeridas num sistema de processamento de eventos, funcionalidades essas exercidas pelas EPAs e EPNs apresentadas na *logical view*. A Figura 5 apresenta uma visão geral da *functional view*. As funcionalidades apresentadas nessa *views* são

divididas em três categorias: *run time functions*, *design time functions* e *administration functions*.

Design time functions representam funcionalidades referentes à definição de eventos e de padrões de eventos que devem ser avaliados. Durante a etapa de *design* de um sistema de processamento de eventos, devem ser identificados os eventos e padrões de eventos relevantes para o sistema em questão. Deve ser especificado como os eventos compostos (complexos) devem ser originados (*i.e.*, a partir de quais padrões e de quais operações), bem como o comportamento esperado dos eventos no sistema, como são manipulados e modelados. As ações a serem executadas pelo sistema quando um determinado padrão de eventos é avaliado como verdadeiro também devem ser definidas durante essa etapa.

Run time functions representam as funcionalidades referentes à produção, processamento e consumo de eventos. Essas funcionalidades concentram a maior parte do processamento de sistemas de eventos. Na camada inferior dessa *view* eventos são produzidos e consumidos. As camadas subsequentes representam funcionalidades executadas entre a produção de um evento e o consumo do mesmo, subdivididas em: *Event Preparation*, *Event Analysis*, *Complex Event Detection* e *Event Reaction*. É importante observar uma cardinalidade “0..*” indicada em cada camada de funcionalidades, indicando que nem todas as camadas são obrigatórias e cada uma pode ser apresentada mais de uma vez no mesmo sistema. Cada camada, conforme apresentado pela Figura 5, é subdividida em outras funcionalidades, como o grupo *Event Preparation*, subdividido em *Identification*, *Selection*, *Filtering*, *Monitoring* e *Enrichment*. Além disso, apesar dessas camadas apresentarem certo grau de ordenação lógica, sua ordem também não é obrigatória, podendo variar de sistema para sistema. Cada camada é brevemente explicada a seguir:

***Event Preparation*:** Antes de qualquer análise sobre os eventos, é importante detectar quando diferentes eventos dizem respeito às mesmas entidades (produto, localização, pessoa, etc). Isso permite uma melhor acurácia nas próximas etapas, sendo necessário algum mecanismo de *tag* de forma que eventos que se refiram às mesmas entidades recebam os mesmos identificadores. Essa etapa é representada pela funcionalidade *Identification* na Figura 5. Após isso, os eventos recebidos podem ser selecionados (*Selection*) e filtrados (*Filtering*) com base em alguma

informação contida nos eventos, por exemplo, *metadados*, de forma a selecionar um subconjunto dos eventos que devem ser processados. Nessa etapa, eventos podem ser ainda convertidos para outro formato, compatível com etapas posteriores ou um formato específico do consumidor. Eventos podem ser também enriquecidos (*Enrichment*) a partir de dados de outras fontes (como bancos de dados) ou de outros eventos.

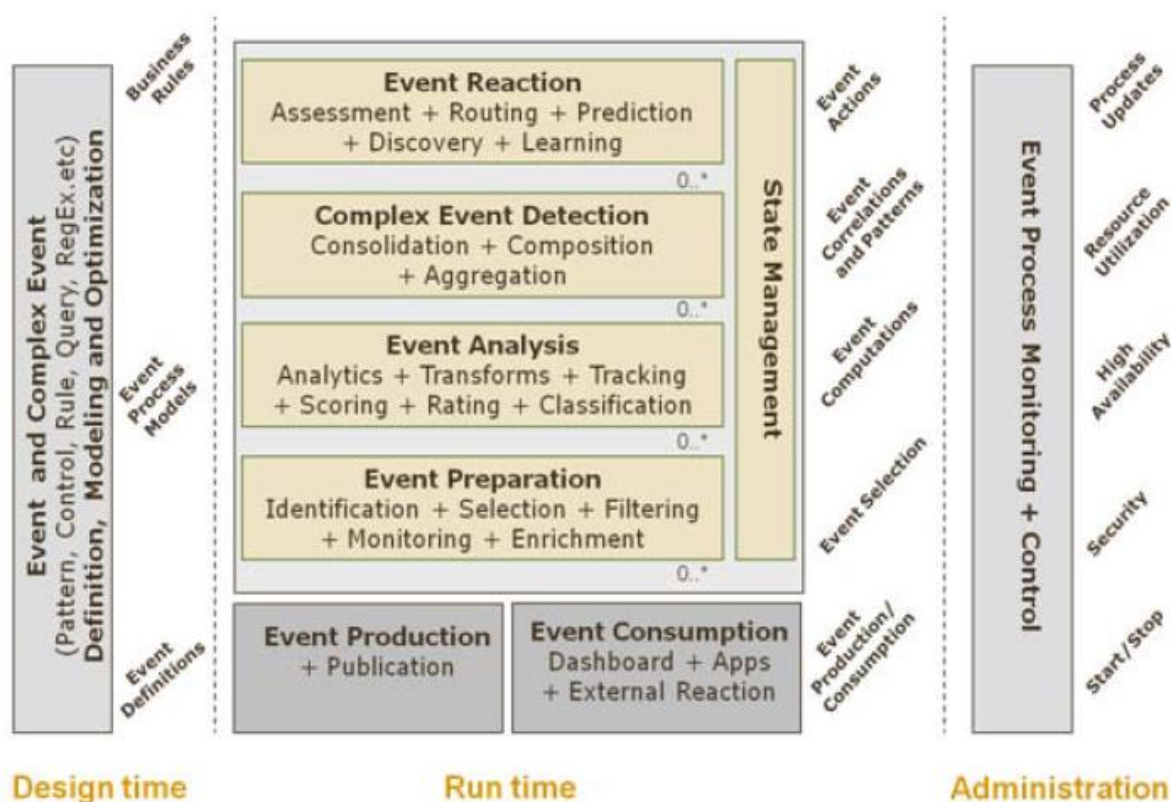


Figura 5. *Functional view* da arquitetura de referência proposta pela EPTS (Paschke, Vincent, Alves, & Moxey, 2012).

Event Analysis: Técnicas de análise (*Analytics*) podem ser aplicadas com o intuito de executar predição sobre os eventos recebidos, por exemplo, analisar alguma tendência no comportamento dos eventos. Transformações (*Transformation*) são possíveis, de forma a converter os dados para algum formato interno (semelhante ao que ocorre na etapa anterior, mas para um formato próprio do sistema) ou normalizar os dados recebidos. Mecanismos de rastreamento (*Tracking*) de eventos pelo sistema podem ser incorporados. Além disso, eventos podem ser pontuados (*Scoring*) e classificados (*Rating*) por meio dos valores e associações

impostas sobre os eventos. Por fim, podem ser realizadas classificações (*Classification*) de forma a identificar os tipos dos eventos e suas associações.

Event Detection: Essa camada tem como objetivo derivar eventos (originando eventos complexos). Eventos podem ser derivados a partir de agregação (*Aggregation*), em que um conjunto de eventos é combinado em um único evento; consolidação (*Consolidation*), em que novos dados são adicionados aos eventos; e composição (*Composition*) em que eventos novos são gerados baseados nas informações contidas em um ou mais eventos (e não pela combinação direta desses eventos, como ocorre em uma agregação).

Event Reaction: Essa camada identifica as ações que devem ser tomadas a partir dos eventos processados, normalmente baseadas nos eventos complexos derivados da etapa anterior (*Event Detection*). Essas ações incluem avaliar (*Assessment*) mudanças de estados do sistema; encaminhar (*Routing*) eventos para seu destino correspondente; predição de futuros comportamentos (*Prediction*); aprendizagem (*Learning*), utilizando técnicas de *machine learning* ou; a descoberta (*Discovery*) de novos tipos de eventos ou padrões.

Por fim, *administrative functions* envolvem funcionalidades referentes ao monitoramento do sistema de processamento de eventos, em tempo de execução, garantindo o desempenho e disponibilidades esperadas, controlando a utilização de recursos, atualizando o sistema e gerenciando questões relacionadas à segurança do sistema.

2.2.2 Event Processing Language (EPL)

Uma EPL (*event processing language*) é uma linguagem computacional que descreve, de forma precisa, padrões de eventos de interesse (Luckham D. , 2002). Quando um padrão definido por uma EPL é satisfeito, dizemos que ocorreu um *match*. Sistemas de CEP utilizam EPLs para avaliar, de forma contínua, os fluxos de eventos utilizados como entrada nos sistemas de CEP, a fim de encontrar padrões de interesse sobre os eventos recebidos. Segundo (Luckham D. , 2002), uma EPL deve apresentar as seguintes propriedades:

1. **Expressividade:** Uma EPL deve permitir a definição de padrões complexos, relacionando eventos por meio de operadores relacionais (ex: *and*, *or*), operadores temporais (ex: *during*, *at*), além de prover mecanismos para acessar dados internos dos eventos;
2. **Simplicidade de notação:** Uma EPL deve permitir a definição de padrões de forma simples e sucinta;
3. **Semântica precisa:** Devem existir conceitos matematicamente precisos de *match*, de forma que ao definir um padrão seja conhecido a priori o conjunto de eventos esperados como saída e;
4. **Match escalável:** Os mecanismos responsáveis por avaliar o *match* dos padrões devem ser escaláveis, processando de forma eficiente uma enorme quantidade de padrões em tempo real, uma vez que sistemas de processamento de eventos lidam com uma enorme quantidade de dados.

Essas propriedades têm grande influência sobre o *design* de uma EPL. Semântica precisa, simplicidade de notação e *match* escalável apresentam requisitos conflitantes em relação à expressividade. Uma EPL simples e fácil de usar não permite a definição de padrões complexos. Por outro lado, uma EPL com alta expressividade, permitindo a definição de padrões complexos, contém recursos avançados nos quais é necessário passar por uma curva de aprendizado para usufruir de todas as funcionalidades disponíveis. Além disso, padrões complexos demandam processamentos complexos nos quais apresentam dificuldades ao implementá-los de forma eficiente (Luckham D. , 2002).

(Fülöp, et al., 2010) defende a classificação de uma EPL em três possíveis estilos:

- 1) **Operadores de composição:** EPLs que permitem expressar eventos complexos a partir da composição de eventos mais simples. Exemplo de abordagem desse estilo é representado por *IBM Active Middleware Technology* (Botzer, 2004);

- 2) **Linguagem de consulta em fluxos de dados:** O estilo mais popular atualmente. Baseado em SQL, linguagens de consulta em fluxos de dados (*Data Stream Query Language*) convertem as tuplas contidas nas *event streams* em relações de bancos de dados. Posteriormente, *queries* SQL são avaliadas sobre esses dados, sendo então convertidos novamente em tuplas. Exemplos de abordagens que fazem uso desse estilo são representados por Coral8 (Coral8, 2009) e Esper (Esper, 2006);
- 3) **Regras de produção:** Especificam ações que devem ser executadas quando certas condições são verificadas. Apesar de não representar uma linguagem propriamente dita, sua utilização é compatível com sistemas de CEP. Exemplo de abordagem desse estilo é representado por *TIBCO Business Events* (TIBCO, 2015).

2.3 SITUAÇÕES

“Contexto é qualquer informação que possa ser usada para caracterizar a situação de uma entidade. Uma entidade é uma pessoa, lugar ou objeto considerado relevante para a interação entre usuário e uma aplicação, incluindo o próprio usuário e a própria aplicação” (Abowd & Dey, 1999). Em outras palavras, toda informação utilizada para caracterizar um participante de uma interação entre usuários e aplicações é dita contexto. Exemplos de contexto são dados pela localização de um usuário ou sua respectiva temperatura corporal.

Aplicações que fazem uso de informações contextuais são ditas aplicações sensíveis ao contexto (*context-aware applications*) (Costa, 2007). Tipicamente, aplicações sensíveis a contexto obtêm informações sobre o contexto do ambiente por meio de sensores (de temperatura, localização, luminosidade, etc.). Informações de contexto também podem ser obtidas por outros meios, sendo informados pelo usuário, por exemplo. Identificar informações de contexto relevantes consiste em determinar o estado de entidades do domínio que são relevantes para uma ou mais aplicações sensíveis ao contexto. (Costa, 2007) classifica contextos em duas

categorias: (i) contexto intrínseco (*Intrinsic Context*) e; (ii) contexto relacional (*Relational Context*).

Contextos intrínsecos dizem respeito a uma única entidade, estando intimamente ligado à sua essência. O contexto intrínseco por si só não existe separado de uma entidade. Além disso, o contexto intrínseco independe da relação de sua entidade com demais entidades do domínio. A localização e temperatura de uma pessoa, são exemplos de contextos intrínsecos. Contextos relacionais, ao contrário de contextos intrínsecos, dizem respeito a mais de uma entidade, representando a relação existente entre os mesmos. Um exemplo de contexto relacional pode ser dado por “DispositivosDisponíveis”. Esse contexto relaciona uma pessoa com uma coleção de dispositivos disponíveis para a pessoa em questão.

Conforme discutido na seção 1.1, aplicações sensíveis ao contexto, geralmente, estão interessadas não apenas nos valores associados ao contexto, mas no significado que aquele valor representa. Esse “significado” representa um estado de interesse da aplicação sensível a contexto, e é denominado *Situação*. (Costa, 2007). Situações são entidades compostas cujos constituintes são outras entidades, suas propriedades e as relações em que estão envolvidos. Exemplos de situações são: “João está com febre”, “João teve uma febre intermitente durante os últimos seis meses”, etc. É possível não só identificar situações, mas também referir-se às propriedades das situações (Barwise, 1989) (Costa, 2006) (Kokar, Matheus, & Baclawski, 2009). Por meio das propriedades (atributos) de uma situação particular podemos, por exemplo, referenciar sua duração ou verificar se a situação é uma situação atual ou passada.

Tipos de situação (*situation type*) (Kokar, Matheus, & Baclawski, 2009) (Costa, 2007) permitem considerar as características gerais de situações de um tipo particular, capturando os critérios de unidade e identidade de situações desse tipo. Um mesmo tipo de situação pode ser instanciado múltiplas vezes. Além disso, um tipo de situação pode ser definido em termos de tipos de situação mais simples. Nesse caso, dizemos que o tipo de situação é um tipo complexo de situação. Um exemplo um tipo de situação é “paciente está com febre”. Instâncias desse tipo de situação são criadas à medida que instâncias de “paciente” (“João”, “Paulo”, etc.)

estejam na condição de “está com febre”, por exemplo, a instância “João está com febre”.

Os exemplos acima revelam a necessidade de se referir a *entity types* como parte da descrição de um tipo de situação. Para o tipo de situação “paciente está com febre”, dizemos que “paciente” é um *entity type*, enquanto que “está com febre” é definida em termos de uma propriedade do *entity type* “paciente” (*i.e.*, a temperatura corporal do paciente).

O processo de detecção de situações (*i.e.*, instâncias de um tipo de situação) consiste, então, em detectar instâncias dos *entity types* envolvidos na situação cujas propriedades satisfazem as restrições capturadas no tipo de situação. Enquanto as restrições capturadas no tipo de situação forem satisfeitas pelas propriedades dos *entity types* envolvidos na situação, dizemos que a situação está *ativa*. A partir do momento em que uma das propriedades dos *entity types*, envolvidos numa situação ativa, não mais satisfaz alguma das restrições definidas pelo tipo de situação, essa situação deixa de existir e, nesse caso, é dita uma situação *passada*.

Uma situação está associada a um intervalo de tempo. A existência de um intervalo de tempo garante um *ciclo de vida* associado à situação, permitindo relacionar situações de forma temporal. Normalmente, essas relações temporais são realizadas por meio das relações de álgebra de intervalos definidas em (Allen, 1983). A Figura 6 ilustra essas relações, são elas: *before*, *after*, *meets*, *metby*, *overlaps*, *overlappedby*, *finishes*, *finishedby*, *starts*, *startedby* e *coincides*. O eixo horizontal representa a dimensão temporal, enquanto que blocos na cor preta representam situações passadas.


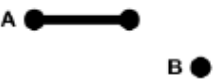

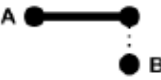



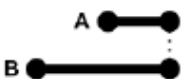





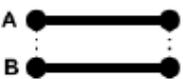
	Point - Point	Point - Interval	Interval - Interval
A Before B B After A			
A Meets B B Met by A			
A Overlaps B B Overlapped by A			
A Finishes B B Finished by A			
A Includes B B During A			
A Starts B B Started by A			
A Coincides B			

Figura 6. Operadores temporais entre situações apresentados em (Pereira et al., 2013).

Uma situação possui um indicador temporal de início e, possivelmente, um indicador temporal de fim. O instante no tempo em que ocorre a detecção de uma instância de uma situação particular (*i.e.*, quando a situação é ativada) é denominado *situation activation instant*. Da mesma forma, o instante no tempo em que a situação deixa de existir (*i.e.*, quando a situação é desativada) é denominado *situation deactivation instant*. É importante destacar que uma situação pode permanecer ativa por tempo indeterminado e, por isso, nem sempre é necessário estar vinculado a um indicador temporal de fim.

A Figura 7 ilustra, por meio de um gráfico, o ciclo de vida de um tipo de situação. O eixo vertical representa os possíveis valores de interesse do domínio (por exemplo, a temperatura corporal de um indivíduo). O eixo horizontal representa a passagem do tempo. Nessa figura, é possível observar a ocorrência de três instâncias diferentes. A ocorrência de uma situação é caracterizada quando a grandeza observada encontra-se acima de um limiar (área cinza na Figura 7). Por questões de simplicidade, a Figura 7 apresenta a variação de apenas uma

propriedade. Um possível exemplo representado pelo gráfico da Figura 7 consiste na temperatura corporal de uma instância (ex. João) do *entity type* “paciente”, referente ao tipo de situação “paciente está com febre”.

Com base nas características de situações apresentadas, é possível destacar os seguintes requisitos básicos para uma abordagem baseada em situações:

1. Tipos de situação devem ser definidos em tempo de projeto, e suas instâncias devem ser detectadas em tempo de execução;
2. Tipos de situação devem ser definidos em relação às entidades, bem como as restrições sobre as propriedades e relações dessas entidades;
3. As propriedades temporais das situações devem ser consideradas (ex. tempo inicial).

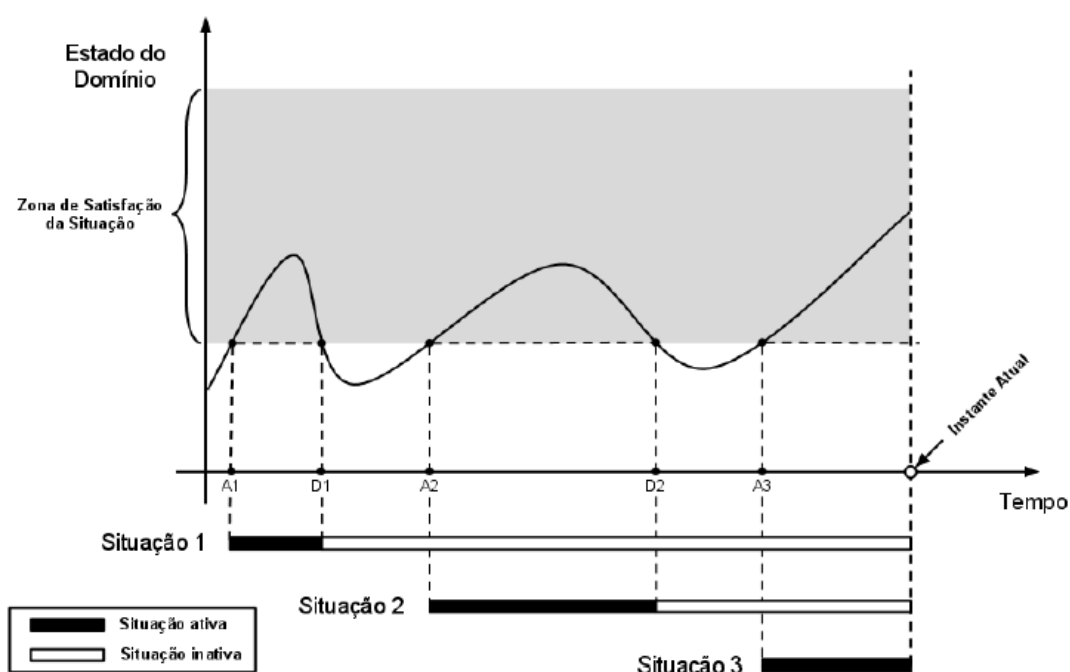


Figura 7. Exemplo do ciclo de vida de um tipo de situação (Pereira, 2013)

2.4 SML

Conforme discutido na seção 1.3, a fim de prover suporte para a especificação de situações em tempo de projeto, baseado no objetivo 2, o presente trabalho

utilizou a linguagem SML (*Situation Modeling Language*) (Mielke, 2013) (Costa, 2012). SML consiste numa linguagem gráfica para a modelagem de situações, apresentando construções que permitem representar situações simples e complexas, num alto nível de abstração. Além disso, SML possui um editor gráfico, disponibilizado na forma de um *plug-in* para Eclipse², desenvolvido com base na ferramenta *ObeoDesigner* (discutido na seção 6.4.1), voltado para a modelagem de situações, facilitando a definição de tipos de situação em tempo de projeto.

A definição de um tipo de situação em SML é composta por dois tipos de modelos: modelos de contexto e modelos de situações. Modelos de contexto definem as entidades e relacionamentos existentes no domínio, enquanto modelos de situações definem os tipos de situações do domínio na forma de padrões gerados a partir das entidades e relações especificadas no modelo de contexto.

A Figura 8 ilustra um exemplo simples de um modelo de contexto. Nesse modelo, é possível observar a entidade Paciente (*Patient*), possuindo atributos (*i.e.*, informações contextuais) sintoma (*symptom*), temperatura (*temperature*) e precisaDeHospitalização (*needHospitalization*). Todo Paciente é também uma Pessoa (*Person*), que possui seu respectivo atributo nome (*name*). Cada Pessoa mora em uma Cidade (*City*), que possui os atributos longitude, latitude e nome. A relação entre Pessoa e Cidade é representada pelo Contexto Relacional “*Lives*”, através das associações *isLiving* e *isInhabited*.

A partir do modelo de contexto apresentado pela Figura 8, considere a definição de três tipos de situação:

1. Situação “Febre” (*Fever*): ocorre se a temperatura de um paciente for superior a 37 graus;
2. Situação “Mora Em” (*LivingIn*): ocorre a partir da existência de uma relação *Lives* entre uma Pessoa e uma Cidade;
3. Situação “Tosse” (*Cough*): ocorre se um paciente possui o sintoma de tosse;

² <https://eclipse.org/>

4. Situação “ILI”: ocorre para todo paciente na situação febre e tosse por 10 dias;

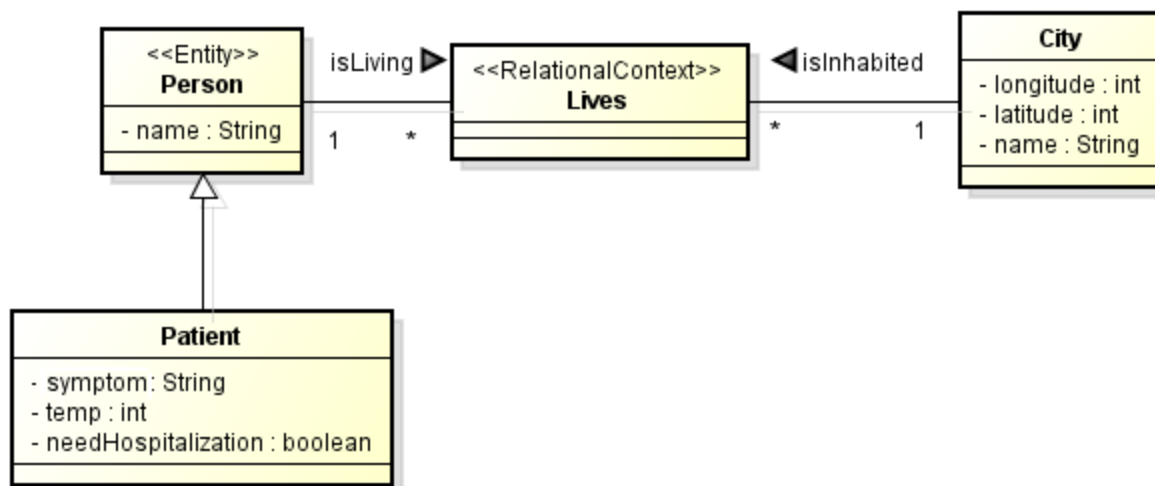


Figura 8. Exemplo de modelo de contexto

O tipo de situação Febre, cuja especificação em SML está ilustrada na Figura 9, assim como o tipo de situação *LivingIn* (Figura 10), representam tipos de situação simples. O tipo de situação ILI, por sua vez, representa um tipo de situação composto, definido a partir de dois tipos de situações simples: Febre e Tosse. A Figura 11 ilustra a especificação desse tipo de situação em SML. Como o diagrama de situação para o tipo Tosse é análogo ao tipo de situação Febre, por simplicidade, o mesmo foi omitido.

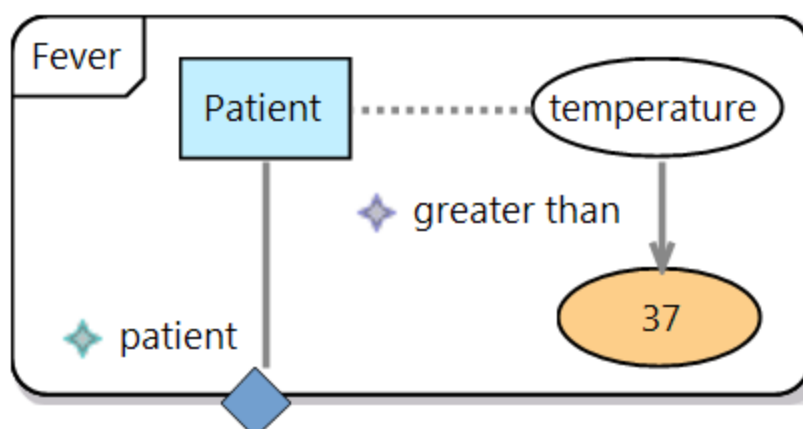


Figura 9. Exemplo de diagrama SML referente ao tipo de situação *Fever*

O retângulo maior, com bordas arredondadas, representa a especificação de um tipo de situação, composta por participantes do padrão, bem como suas relações e atributos. Em SML um participante pode ser uma Entidade, uma Situação ou um *Relator* (representando a ocorrência de um contexto relacional). Por sua vez, o retângulo menor representa uma entidade. Tomemos por exemplo a situação Febre (Figura 9). A entidade apresentada corresponde ao tipo Paciente (*Patient*). Paciente está associado a um atributo temperatura (*temperature*), representado por uma elipse branca. Esse atributo, por sua vez, está associado a um literal, representado por uma elipse preenchida, de valor igual a 37. O atributo e o seu respectivo valor são relacionados por meio de uma seta. A seta direcional representa a ocorrência de uma relação formal.

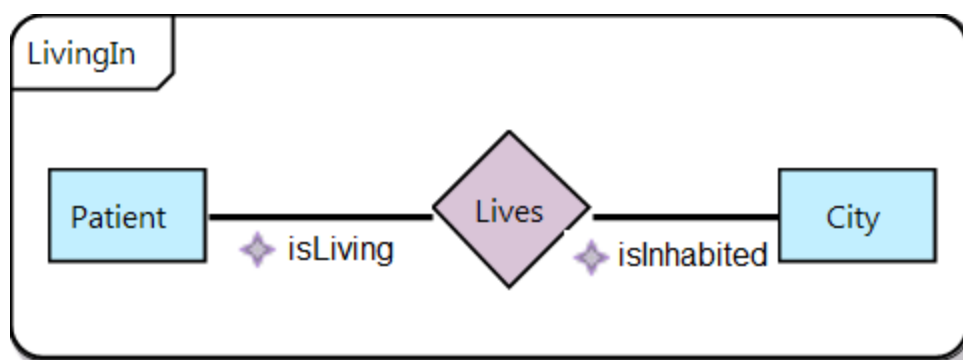


Figura 10. Exemplo de diagrama SML referente ao tipo de situação *LivingIn*

Na situação Febre, a relação formal utilizada corresponde à relação “maior que” (*greater than*). Relações formais são aplicadas diretamente sobre elementos participantes de um tipo de situação. Algumas relações formais encontram-se definidas pela linguagem SML, por exemplo, as relações “maior que” (*greater than*), “menor que” (*less than*) e “igual” (*equals*). Relações formais que não encontram-se definidas na linguagem SML podem ser especificadas pelo usuário no modelo de contexto.

Dessa forma, o diagrama referente à situação Febre (Figura 9) descreve que a mesma ocorre quando uma entidade do tipo Paciente possui o atributo temperatura com valor maior que 37. Além disso, é possível ver que a entidade Paciente possui uma associação denominada *patient*, conectada à borda da situação por meio de um losango. Essa relação indica que a entidade Paciente envolvida na situação pode ser referenciada por outras situações por meio do nome *patient*.

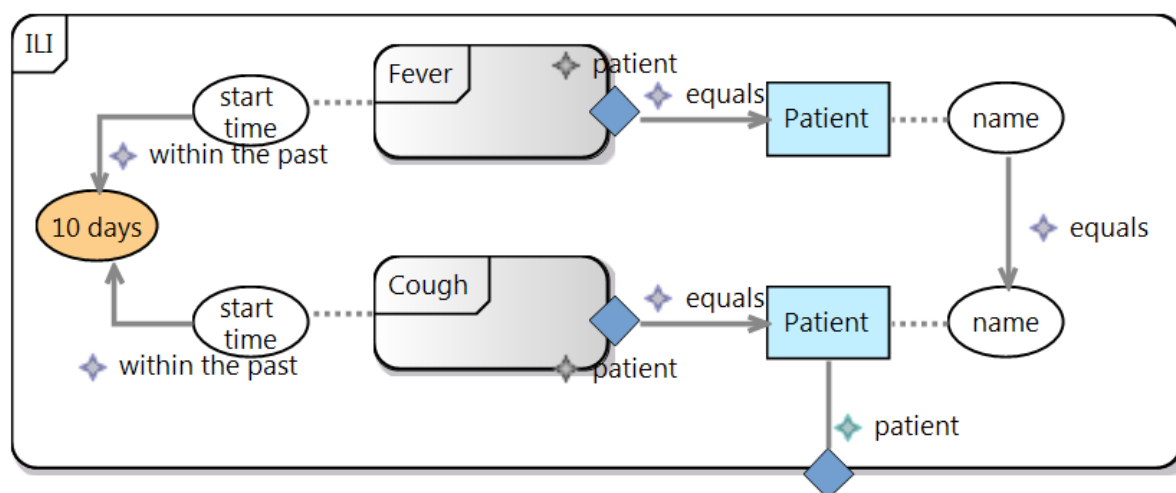


Figura 11. Exemplo de diagrama SML referente ao tipo de situação ILI

O tipo de situação *LivingIn* (Figura 10), ilustra a utilização de um *Relator* (contexto relacional), representado por um losango colorido na cor roxa. Um *Relator* relaciona as entidades por meio de linhas, sem direção, coloridas na cor preta. Na Figura 10, as relações *isLiving* e *isInhabited* são utilizadas para relacionar as entidades dos tipos Paciente e Cidade, respectivamente. Assim, o tipo de situação *LivingIn* descreve que a mesma ocorre sempre que existir um contexto relacional *Lives*, relacionando um Paciente a uma Cidade.

Por sua vez, o tipo de situação ILI (Figura 11), conforme discutido anteriormente, é composta por tipos de situações mais simples: Febre e Tosse. Ambos os tipos de situações são representados por um retângulo menor, de bordas arredondadas. Além disso, os tipos de situação Febre e Tosse estão associados por uma janela temporal, representada pela relação formal “*within the past*” e o atributo tempo inicial (*start_time*), presente em toda situação. Essa relação indica que a situação deve estar ativa pelos últimos 10 dias. Dessa forma, o tipo de situação ILI especifica que a mesma deve ocorrer quando um mesmo Paciente encontra-se envolvido nas situações Febre e Tosse, durante 10 dias.

A Figura 12 apresenta o exemplo da ocorrência no tempo referente a uma instância do tipo de situação ILI. Inicialmente uma instância do tipo de situação Febre é ativada. No momento em que uma instância do tipo de situação Tosse é ativada, enquanto a instância do tipo de situação do tipo Febre mantém-se ativa, respeitando a restrição de 10 dias, é ativada também uma instância do tipo de

situação ILI. De forma análoga, no momento em que uma das duas instâncias do tipo Febre ou do tipo Tosse é desativada, a instância correspondente ao tipo ILI é também desativada. A especificação completa da linguagem SML pode ser encontrada em (Mielke, 2013).

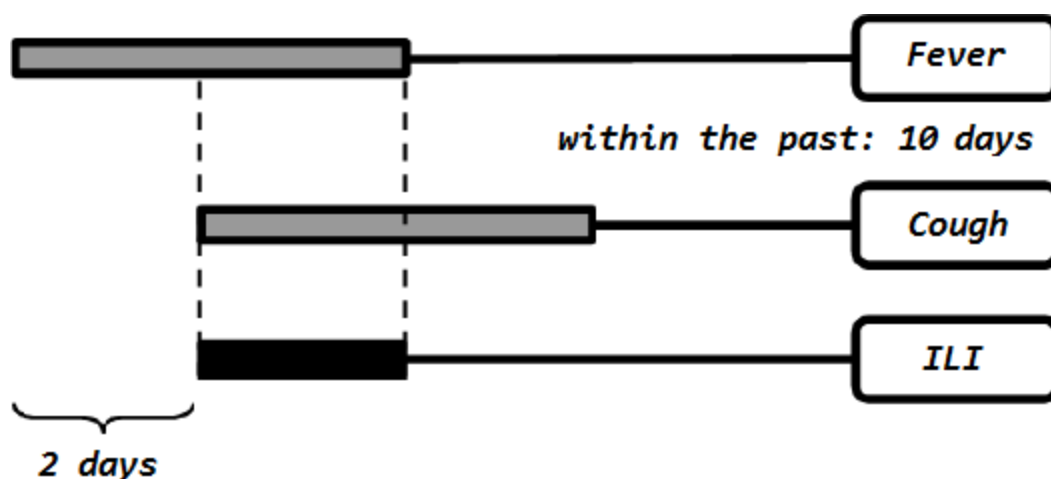


Figura 12. Exemplo de ocorrência no tempo de uma instância do tipo de situação ILI

2.5 DESENVOLVIMENTO ORIENTADO A MODELOS

Abordagem de Desenvolvimento Orientado a Modelos (MDD) (Almeida, Pires, & Sinderen, 2004) (Almeida J. , 2006) “é um estilo de desenvolvimento de *software* onde os artefatos primários são modelos nos quais são gerados códigos e outros artefatos” (Swithinbank, et al., 2005). Modelos, por sua vez, representam “uma descrição do sistema de uma perspectiva particular, omitindo detalhes irrelevantes, permitindo que características de interesse sejam observadas de forma mais clara” (Swithinbank, et al., 2005). De acordo com essas definições, MDD é um estilo de desenvolvimento de *software* que permite o desenvolvimento num nível maior de abstração através de modelos, nos quais são utilizados para gerar outros artefatos.

A utilização de modelos como artefato primário permite a utilização de conceitos menos tecnológicos e mais próximos aos conceitos do domínio, ao contrário do que ocorre na maioria das linguagens de programação (Selic, 2003). Modelos utilizados em MDD devem ser *machine-readable* (i.e., devem estar em formatos nos quais podem ser entendidos por computadores), de forma que seja

possível acessar seu conteúdo de forma automatizada. Geralmente, modelos são descritos utilizando UML (*Unified Modeling Language*) (OMG, 2012), uma linguagem padrão para modelagem orientada a modelos, *machine-readable*, que permite a especificação, visualização e documentação de sistemas de *software*.

MDD tem como vantagem diminuir o custo de desenvolvimento e aumentar a consistência e qualidade das soluções (Swithinbank, et al., 2005). Isso é alcançado através da implementação de transformações, nos quais eliminam tarefas repetitivas de menor nível de abstração. Em vez de aplicar inúmeras vezes, de forma manual, conhecimento técnico em tarefas que se repetem ao longo do processo de construção de um artefato (ex: código, modelo, documentação, etc), tal conhecimento pode ser codificado diretamente em transformações, automatizando tais tarefas. Isso permite também melhor manutenibilidade: modificações nos modelos ou transformações rapidamente são refletidas no artefato gerado.

O conhecimento técnico necessário para desenvolvimento dos artefatos é capturado diretamente pelas transformações, permitindo que desenvolvedores abstraia quaisquer aspectos tecnológicos envolvidos e concentrem-se em aspectos de maior nível de abstração. Em alguns casos, é possível que o sistema seja desenvolvido inteiramente por especialistas do domínio, em vez de especialistas de tecnologias de computação (Selic, 2003).

2.6 TRABALHOS RELACIONADOS

2.6.1 Situation Specification and Realization in Rule-Based Context-aware Applications

Em (Costa, 2007) é proposta uma abordagem voltada para a especificação e detecção de situações em aplicações *context-aware*. O objetivo da abordagem consiste em prover abstrações que facilitem a especificação de situações, em tempo de projeto, por meio de diagramas de classe UML juntamente com a utilização de restrições especificadas por meio do padrão OCL. Situações especificadas são

transformados em um conjunto de regras correspondentes, executadas diretamente numa plataforma baseada em regras. Conceitos fundamentais para a elaboração deste trabalho são definidos em (Costa, 2007), por exemplo, as definições de contexto (*context*), entidade (*entity*) e tipo de situação (*situation type*).

Contexto é definido em (Costa, 2007) como “o conjunto de condições (possivelmente relacionadas) nas quais uma entidade existe”. Essa definição sugere que contexto apenas possui significado com respeito a algo que exista. O conceito de entidade é utilizado para representar “algo que exista”. Esse conceito é fundamental para o entendimento de contexto. (Costa, 2007) indica que “contexto é algo que pode ser dito a respeito de uma entidade”. Dessa forma, o contexto não existe por si só, estando sempre relacionado a pelo menos uma entidade. Exemplos de entidades são representados por pessoas, construções, dispositivos, etc. Por exemplo, o contexto pode ser representado pela respectiva localização e temperatura de uma pessoa.

(Costa, 2007), define o conceito de situação apresentado na seção 2.3, conceito este fundamental para o entendimento e desenvolvimento do presente trabalho. A abordagem apresentada em (Costa, 2007) baseia-se na especificação de restrições OCL, uma linguagem de propósito geral, para a definição dos tipos de situações. A utilização de OCL conduz à elaboração de modelos de situação complexos, além de exigir amplo conhecimento dos desenvolvedores. Diferentemente de (Costa, 2007), este trabalho tem como objetivo gerenciar situações em sistemas baseados em eventos, mais precisamente, em sistemas de processamento de eventos complexos (CEP). Além disso, os modelos de tipos de situação são especificados utilizando a linguagem SML (seção 2.4), que foi desenvolvida exclusivamente para a especificação de tipos de situações de forma independente de tecnologia ou plataforma, gerando modelos mais compactos e possibilitando a especificação de situações num nível maior de abstração.

2.6.2 DS-EPL: Domain-Specific Event Processing Language

(Bruns, Dunkel, Lier, & Masbruch, 2014) propõem a aplicação de conceitos relacionados à DLS (*Domain Specific-Language*), na área de CEP (*Complex Event Processing*), com o intuito de desenvolver DS-EPLs (*Domain-Specific Event Processing Languages*) voltadas à solução de problemas particulares do domínio das aplicações CEP. Por meio de uma linguagem DS-EPL, desenvolvedores são capazes de especificar padrões EPL (*Event Processing Language*) utilizando elementos específicos do domínio, permitindo que desenvolvedores especifiquem tais padrões num nível mais alto de abstração. De forma semelhante ao que ocorre neste trabalho, o código gerado pela linguagem DS-EPL é transformado para código específico da plataforma alvo utilizada. Para validar sua abordagem, também foi utilizada a plataforma Esper.

A abordagem proposta por (Bruns, Dunkel, Lier, & Masbruch, 2014) assemelha-se à adotada por este trabalho nos seguintes aspectos: elementos do domínio são mapeados em modelos de forma similar aos modelos de contexto e situação, permitindo que a linguagem gerada utilize conceitos relacionados ao domínio modelado. Porém, ainda que a DS-EPL permita ao desenvolvedor trabalhar num nível mais alto de abstração, o mesmo ainda deve conhecer as regras, padrões e demais restrições que compõem a linguagem DS-EPL gerada. SML também é uma linguagem específica de domínio, porém permite que o usuário modele situações de forma gráfica, totalmente independente de tecnologia. Além disso, a abordagem proposta em (Bruns, Dunkel, Lier, & Masbruch, 2014) provê suporte apenas para especificação de regras em tempo de projeto, não abordando aspectos relacionados à implementação de situações, por exemplo, detecção e gerenciamento do ciclo de vida. Por fim, o aspecto de composicionalidade das situações também não é explorado.

2.6.3 Learning from the past: automated rule generation for complex event processing

Em (Margara, Cugola, & Tamburrelli, 2014) é apresentado um *framework* denominado iCEP, que tem como objeto a geração automática de regras para uma plataforma CEP, a partir de inferências sobre os fluxos de eventos. iCEP analisa históricos de eventos recebidos, aplicando algoritmos de aprendizagem sobre os mesmos.

Esse *framework* possui sete módulos, onde cada módulo concentra-se em um aspecto diferente das regras, sendo estes responsáveis por: (i) determinar o tamanho das janelas; (ii) identificar atributos e tipos de eventos relevantes; (iii) determinar restrições sobre os atributos; (iv) determinar restrições sobre seleções; (v) identificar sequências; (vi) identificar agregações e; (vii) identificar negações. O usuário pode decidir, baseado em seu conhecimento do domínio, quais módulos deseja utilizar, evitando processamento desnecessário. Por exemplo, caso o domínio em questão não abranja padrões que utilizem de negações, o usuário pode desativar o módulo correspondente.

A falta de suporte para a especificação de regras nas plataformas de CEP é abordada em (Margara, Cugola, & Tamburrelli, 2014), que enfatiza que grande partes dos trabalhos na área de CEP focam no processamento eficiente dos fluxos de eventos. Este aspecto é importante e depende de uma correta e precisa modelagem do domínio da aplicação. Porém, pouco se aborda quanto aos aspectos referentes à modelagem do domínio da aplicação. As plataformas desenvolvidas são capazes de detectar padrões complexos, porém, o usuário não recebe suporte adequado para especificar tais padrões. Padrões simples do mundo real podem tornar-se extremamente complexos quando representados pelas linguagens EPL (*Event Processing Language*). Essa tarefa torna-se ainda mais árdua considerando que, por vezes, o especialista do domínio não possui conhecimento sobre as plataformas utilizadas e, consequentemente, sobre as linguagens EPL utilizadas.

Apesar do trabalho apresentado em (Margara, Cugola, & Tamburrelli, 2014) diferir do presente trabalho, uma vez que visa a geração automática de regras e não a detecção e gerenciamento das mesmas, ambos possuem objetivos comuns.

Ambos propõem abordagens que visam facilitar a modelagem de regras por parte do usuário, utilizando técnicas distintas: (Margara, Cugola, & Tamburrelli, 2014) utiliza um *framework* que gera regras automaticamente por meio de técnicas de aprendizagem, enquanto o presente trabalho utiliza uma abordagem de desenvolvimento orientada a modelos.

2.6.4 Situation-Aware Energy Control by Combining Simple Sensors and Complex Event Processing

Em (Renner, Bruns, & Dunkel, 2012) é apresentada uma abordagem para controle inteligente de energia, que combina uso de sensores de baixo custo com processamento de fluxos de eventos capturados por meio desses sensores, permitindo a detecção de situações de alto nível. Para validar a abordagem, foram utilizados ambientes de salas de uma universidade. Dentre os objetivos traçados, destacam-se quatro: (i) controle individual de cada sala; (ii) infraestrutura de baixo custo; (iii) ambiente *situation-awareness* e; (iv) controle proativo.

Na abordagem proposta, elementos e eventos do domínio são mapeados em modelos: *Semantic Domain Model* e *Event Model*, respectivamente. Esses modelos funcionam de forma similar aos modelos de contexto e situação, possibilitando a definição de situações em função de elementos específicos do domínio.

Assim como em (Bruns, Dunkel, Lier, & Masbruch, 2014), a abordagem proposta em (Renner, Bruns, & Dunkel, 2012) trata situações apenas com a detecção de padrões pontuais. Ambos os trabalhos visam detectar e especificar situações sem levar em consideração o ciclo de vida associado às situações. Definir e gerir situações utilizando eventos primitivos consiste em uma tarefa complexa, principalmente quando se trata de situações com alto nível de composição (situações complexas compostas por demais situações complexas). O presente trabalho defende que a gerência do ciclo de vida das situações deve estar incorporada ao sistema CEP, permitindo ao desenvolvedor trabalhar em um nível mais alto de abstração, definindo situações compostas a partir de situações mais simples, definidas previamente.

2.6.5 Análise Comparativa

Conforme discutido na seção 1.3, a etapa inicial para o desenvolvimento deste trabalho consistiu em um estudo de propostas de plataformas CEP e do uso de situações em aplicações sensíveis ao contexto relatadas na literatura, com o objetivo de obter entendimento a respeito dos conceitos e requisitos necessários para o desenvolvimento de aplicações nessas áreas. Durante esta etapa foram definidas algumas funcionalidades consideradas essenciais para o desenvolvimento de aplicações voltadas para a apoiar a utilização de situações, tanto em tempo de projeto quanto em tempo de execução.

A tabela ilustrada pela Figura 13 apresenta as sete funcionalidades consideradas mais importantes: (i) suporte para a especificação de padrões em um alto nível de abstração; (ii) utilização de uma linguagem gráfica para especificar os padrões de interesse; (iii) suporte para a especificação de situações no nível de implementação; (iv) suporte para a detecção de situações; (v) mecanismo de gerenciamento do ciclo de vida das situações; (vi) permitir a composicionalidade das situações e (vii) geração de código específico de domínio de forma automatizada. As colunas representam os trabalhos relacionados: (Costa, 2007) (Bruns, Dunkel, Lier, & Masbruch, 2014) (Margara, Cugola, & Tamburrelli, 2014) (Renner, Bruns, & Dunkel, 2012).

Funcionalidades	Trabalhos Relacionados			
	(Costa, 2007)	(Bruns, 2014)	(Margara, 2014)	(Renner, 2012)
Suporte para a especificação de padrões num nível maior de abstração	X	X	X	X
Utiliza uma linguagem gráfica para especificar os padrões de interesse				
Especificação de situações	X			X
Detecção de situações				X
Gerenciamento do ciclo de vida das situações				
Composicionalidade das situações	X			
Geração automatizada de código específico de domínio	X	X	X	X

Figura 13. Tabela comparativa dos trabalhos relacionados

Conforme é possível observar na Figura 13, todos os trabalhos relacionados apresentam suporte para a especificação de padrões em um alto nível de abstração. (Costa, 2007), (Bruns, Dunkel, Lier, & Masbruch, 2014) e (Renner, Bruns, & Dunkel, 2012) permitem essa especificação através da utilização de modelos, cada um utilizando um tipo diferente de modelo. Foi considerado que o trabalho apresentado

em (Margara, Cugola, & Tamburrelli, 2014) provê tal funcionalidade, pois, apesar de não permitir que o usuário especifique seus padrões de interesse, estes são gerados automaticamente (permitindo que o usuário concentre-se nos padrões gerados ao invés de lidar diretamente com os fluxos de dados).

Além disso, os quatro trabalhos fazem uso de técnicas que permitem a geração automática de código específico de domínio. (Costa, 2007), (Bruns, Dunkel, Lier, & Masbruch, 2014) e (Renner, Bruns, & Dunkel, 2012) realizam transformações a partir de modelos especificados em um alto nível de abstração, enquanto que (Margara, Cugola, & Tamburrelli, 2014) gera código específico de domínio a partir de inferências realizadas sobre os fluxos de dados recebidos.

Ambos os trabalhos apresentados em (Costa, 2007) e (Renner, Bruns, & Dunkel, 2012) utilizam o conceito de situação, permitindo que o usuário especifique padrões que representem situações. Porém, apenas o trabalho apresentado em (Costa, 2007) considera o aspecto de composicionalidade das situações, conceito este fundamental para permitir a especificação de situações complexas (*i.e.*, situações compostas). O trabalho apresentado em (Renner, Bruns, & Dunkel, 2012) provê suporte para a detecção de situações em tempo de execução. Essa detecção, porém, é realizada de forma limitada, tratando situações apenas como ocorrências pontuais de eventos ao longo do tempo. Conforme o conceito de situação utilizado neste trabalho (seção 2.3) uma situação está associada a um intervalo de tempo, garantindo um *ciclo de vida* associado à situação e permitindo relacionar situações de forma temporal.

Conforme é possível observar na Figura 13, nenhum dos trabalhos apresentados cobre totalmente as sete funcionalidades apresentadas. Além disso, duas funcionalidades, consideradas pelo presente trabalho como imprescindíveis em aplicações voltadas para a utilização de situações, não são consideradas por nenhum dos trabalhos apresentados: (i) a utilização de uma linguagem gráfica para especificar os padrões de interesse e (ii) prover suporte adequado para o gerenciamento do ciclo de vida das situações. Ambos os conceitos são considerados para apoiar a utilização de situações em tempo de projeto e em tempo de execução, respectivamente.

O presente trabalho possui como objetivo propor uma metodologia para auxiliar a especificação, detecção e processamento de situações, tanto em tempo de projeto quanto em tempo de execução, levando em conta todas as funcionalidades levantadas durante a etapa de levantamento de requisitos e apresentadas na Figura 13, ao contrário do que foi encontrado nos trabalhos apresentados onde apenas algumas funcionalidades eram disponibilizadas.

2.7 CONSIDERAÇÕES DO CAPÍTULO

Este capítulo apresentou conceitos básicos das áreas de aplicações sensíveis ao contexto, aplicações baseadas em eventos (CEP) e desenvolvimento orientado a modelos. Estes são conceitos fundamentais para o entendimento deste trabalho, que visa propor uma metodologia, orientada a modelos, que ofereça suporte para a detecção e especificação de situações em sistemas orientados a eventos. Foram apresentados e discutidos trabalhos relacionados e, além disso, tais trabalhos foram comparados e analisados com base nas principais funcionalidades, propostas durante a etapa de levantamento de requisitos, desejadas em aplicações voltadas para apoiar a utilização de situações. O capítulo seguinte apresenta Esper, uma plataforma CEP utilizada como base para a definição deste trabalho.

3 ESPer

Conforme discutido na seção 1.2, o segundo objetivo do presente trabalho consiste em desenvolver uma plataforma de CEP que forneça suporte para situações em tempo de projeto e em tempo de execução. Para tal, tomou-se como ponto de partida uma plataforma de CEP já existente. Conforme apresentado na seção 1.3, este trabalho selecionou, dentre as diversas plataformas CEP disponíveis, a plataforma Esper.

Diversos motivos levaram a escolha dessa plataforma, dentre eles, podemos citar: (i) consiste numa plataforma de CEP *open source*; (ii) possui documentação bem detalhada e organizada, descrevendo informações necessárias para seu entendimento e utilização, bem como métricas para avaliação de desempenho utilizando tal plataforma; (iii) é distribuída na linguagem Java, uma linguagem amplamente utilizada no qual grande parte dos desenvolvedores encontra-se familiarizada; (iv) é disponibilizada na forma de uma biblioteca, sendo facilmente configurada e incorporada ao ambiente de trabalho e; (v) possui uma EPL altamente expressiva, denominada Esper EPL (seção 3.2), que possui sintaxe baseada na linguagem SQL (também amplamente utilizada), permitindo a definição de padrões complexos (conforme os requisitos apresentados na seção 2.2.2), através de construções que relacionam diferentes fluxos de eventos, por exemplo, agregação, composição e operadores temporais.

Este capítulo tem como objetivo apresentar a plataforma Esper e está organizado da seguinte forma: a Seção 3.1 apresenta uma visão geral da plataforma Esper; a Seção 3.2 apresenta a linguagem utilizada pela plataforma Esper para a especificação de padrões, denominada Esper EPL; a Seção 3.3 apresenta o modelo de processamento de eventos utilizado pela plataforma; 3.4 apresenta o conceito de *event pattern*, bem como as principais construções de padrões utilizadas pelo presente trabalho; 3.5 apresenta um exemplo simples de uma aplicação utilizando a plataforma Esper; e, por fim, a Seção 3.6 apresenta as considerações do capítulo.

3.1 VISÃO GERAL DA PLATAFORMA

Esper³ é uma plataforma CEP, *open source*, distribuída nas linguagens Java e .NET. Esper permite o desenvolvimento de aplicações que lidam com uma grande quantidade de dados, sejam eles históricos ou atuais, provendo requisitos não funcionais como: escalabilidade, uso eficiente de memória e baixa latência. Esper permite a execução de ações escritas em *Plain Old Java Objects* (POJO), que são executadas quando certas condições são verificadas sobre fluxos de eventos. Em Esper, eventos podem ser representados como classes *Java Beans*, classes Java legadas, documentos XML ou `java.util.Map`. O modelo de processamento de eventos (seção 3.3) utilizado pela plataforma Esper consiste na avaliação de forma contínua de padrões aplicados sobre fluxos (infinitos) de eventos. Isso permite processar e avaliar tais eventos no momento de sua ocorrência. Além disso, a plataforma Esper é capaz de lidar com diferentes fluxos de eventos, provendo operações capazes de relacionar os mesmos, como agregações e *joins*.

Padrões são definidos em Esper através de uma *event processing language* própria, denominada Esper EPL (seção 3.2), ou de *event patterns* (seção 3.4). Essa linguagem, baseada em SQL (*Structured Query Language*), permite a especificação de *queries* avaliadas continuamente sobre os fluxos de eventos. A manipulação de eventos em Esper é comparada como um banco de dados invertido: ao invés de armazenar os dados e executar *queries* sobre estes dados, *queries* são armazenadas e executadas sobre fluxos de dados. Além disso, o modelo de execução é contínuo: *queries* são avaliadas sempre que um novo dado é recebido, e não apenas uma vez, como ocorre tradicionalmente em banco de dados.

3.2 ESPER EPL

Esper provê uma *event processing language* (seção 2.2.2), denominada Esper EPL, de forma a permitir a execução de *queries* a serem executadas na plataforma. *Queries* EPL definem padrões que especificam restrições de interesse em fluxos de

³ <http://www.espertech.com/esper/>

eventos. Através da utilização de *queries*, é possível combinar eventos a fim de criar eventos complexos.

A linguagem EPL permite a derivação e agregação de informações a partir de um ou mais fluxos de eventos. Para isso, são disponibilizadas construções que permitem, dentre outras, a especificação de janelas temporais, *join* entre diferentes fluxos de dados, filtros, agregação e ordenação. Essa linguagem é similar à linguagem SQL (*Structured Query Language*), tanto no uso de cláusulas *select* quanto *where*. Porém, ao invés de utilizar tabelas, em EPL existe o conceito de *views*. *Views* são análogas às tabelas SQL, e definem quais dados estão disponíveis para as consultas (*queries*). *Views* podem representar um fluxo completo (infinito) de eventos ou um subconjunto (finito), por exemplo, um subconjunto limitado por uma janela que armazena apenas os últimos cinco eventos recebidos. Além disso, uma *view* permite a aplicação de ordenação (*order by*), agrupamento (*group by*), manipulação de propriedades dos eventos, dentre outras operações, similarmente como ocorre em SQL. A especificação de *views* é realizada por meio da cláusula *from*.

A Figura 14 apresenta o exemplo de uma *query* EPL simples. Essa *query* computa a média (*avg*) de preço (*price*) dos eventos do tipo *StockTickEvent* recebidos nos últimos 30 segundos. Para cada novo evento do tipo *StockTickEvent* recebido pela plataforma Esper, a *query* é reavaliada. Nesse exemplo, a *view* é representada por uma janela temporal, *win:time(30 sec)*, aplicada sobre um fluxo de dados (eventos do tipo *StockStrickEvent*). Para essa *query*, assume-se que existe uma classe Java implementada, cujo nome seja *StockTickEvent* e um dos atributos tenha nome igual à *price*.

```
select avg(price) from StockTickEvent.win:time(30 sec)
```

Figura 14. Exemplo de uma *query* EPL simples

A Figura 15, por sua vez, ilustra uma *query* mais complexa, na qual é possível observar um *join* entre dois fluxos distintos de eventos, *FraudWarningEvent* e *WithdrawalEvent*. Sobre ambos os fluxos de eventos é, novamente, aplicada uma

view apresentada por uma janela temporal de 30 segundos. Os campos retornados pela cláusula *select* não renomeados através do operador “as”. Além disso, existe uma restrição na cláusula *where*, especificando a condição do *join*, nos quais eventos recebidos de ambos os fluxos devem possuir mesmo valor para o atributo *accountNumber*.

```
select fraud.accountNumber as acctNum, fraud.warning as warn, withdraw.amount as amount,
      MAX(fraud.timestamp, withdraw.timestamp) as timestamp, 'withdrawalFraud' as desc
from FraudWarningEvent.win:time(30 min) as fraud,
      WithdrawalEvent.win:time(30 sec) as withdraw
where fraud.accountNumber = withdraw.accountNumber
```

Figura 15. Exemplo de *join* entre dois fluxos diferentes de eventos

3.3 MODELO DE PROCESSAMENTO DE EVENTOS

Conforme dito anteriormente, o modelo de processamento de eventos em Esper é feito de forma contínua sobre os fluxos de eventos. Isso é feito através de *Listeners* ou *Subscribers*, associados às *queries*. Este trabalho optou pela utilização de *Listeners*, e, portanto, apenas tal elemento será explicado, porém tais explicações são válidas ou facilmente adaptadas para a utilização de *Subscribers*.

Apesar do manual da plataforma Esper recomendar a utilização de *Subscribers*, dois motivos levaram a utilização de *Listeners* neste trabalho. O primeiro motivo consiste em uma maior quantidade exemplos e documentação (tanto no manual da plataforma quanto em fóruns) referente ao uso de *Listeners*. O segundo, e principal motivo, consiste no fato dos parâmetros recebidos pelos métodos implementados pelos *Subscribers* serem fortemente tipados, ou seja, é necessário especificar a priori o tipo (e quantidade) exata de cada parâmetro. O presente trabalho optou por desenvolver um componente genérico associado a todas as *queries* referentes a algum tipo de situação, no qual cada *querie* possui um conjunto diferente de eventos. Por este motivo, considerou-se utilização de *Listeners*, no qual os objetos são recebidos pelos seus métodos em uma representação intermediária do tipo *EventBean*, uma melhor escolha ao considerar a utilização de estruturas fortemente tipadas.

Cada *query* EPL deve estar associada a uma classe *Listener* (no padrão POJO). Caso as condições especificadas em alguma *query* sejam satisfeitas por um evento (ou conjunto de eventos), o método *update* correspondente do *Listener* associado à mesma é invocado. Isso permite que as ações apropriadas sejam executadas para cada padrão especificado por meio das *queries* EPL.

Tomemos como exemplo a *query* especificada pela Figura 16. Essa *query* especifica o interesse em todos os eventos do tipo *Withdrawal*, considerando uma janela indicando que os últimos N elementos recebidos devem ser armazenados, no caso, os últimos 5.

```
select * from Withdrawal.win:length(5)
```

Figura 16. Exemplo de *query* utilizando janela

O diagrama ilustrado pela Figura 17 corresponde ao modelo de processamento referente à *query* apresentada pela Figura 16. Esse diagrama apresenta seis eventos do tipo *Withdrawal*, (W1, W2, W3, W4, W5 e W6) recebidos ao longo do tempo. O número entre parênteses representa o valor atributo *amount*, presente em cada evento desse tipo. Esse valor não é utilizado no diagrama em questão, porém é utilizado nos exemplos seguintes.

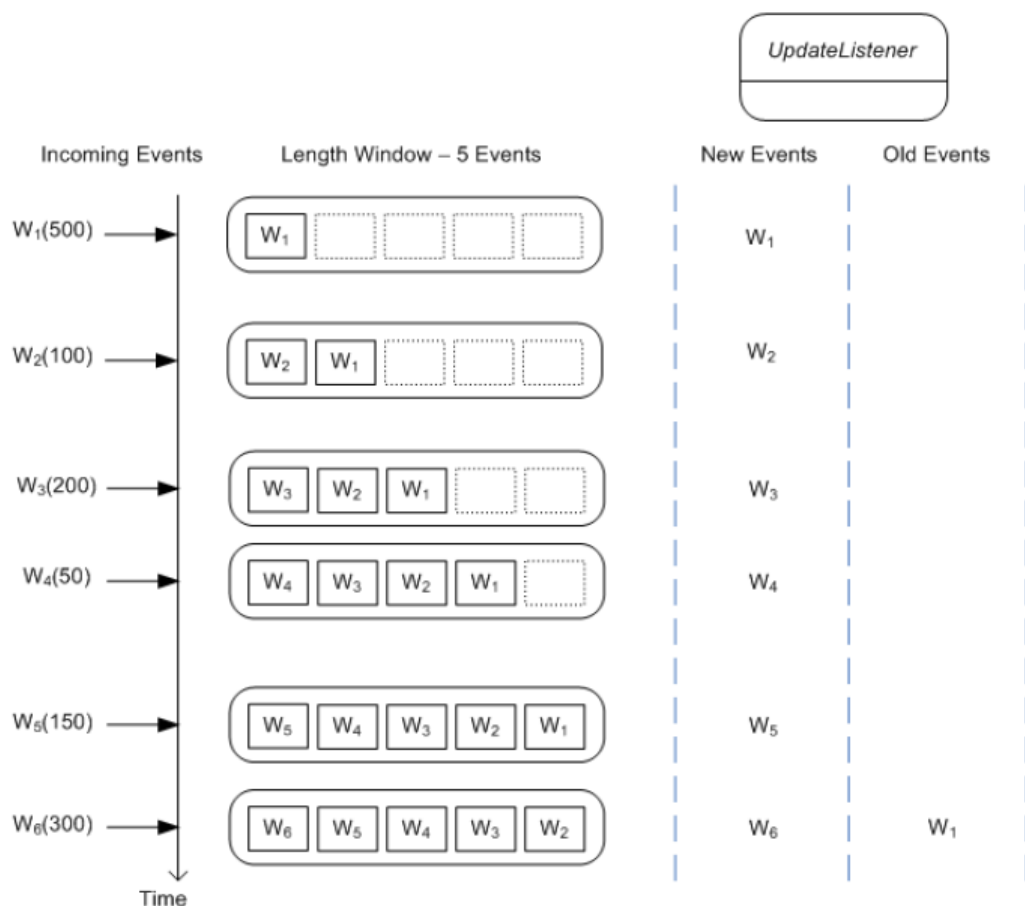


Figura 17. Modelo de processamento utilizando janela

Como não foi especificado nenhum filtro, qualquer evento do tipo *Withdrawal* é inserido na janela. Quando um novo evento é inserido na janela, o método *update* de todos os *Listeners* associados a essa *query* é invocado, enviando o evento do tipo *Withdrawal* correspondente como um novo evento (*new event*). Quando a janela atinge seu tamanho máximo (5 eventos), o evento mais antigo (W_1) é retirado da janela. Nesse momento, os métodos *update* são novamente invocados, porém recebendo o evento do tipo *Withdrawal* correspondente como um evento antigo (*old event*).

O modelo de processamento ao utilizar uma janela temporal ocorre de forma análoga ao apresentado pela Figura 17, porém mantendo na janela os eventos mais recentes considerando um dado período de tempo especificado. A Figura 18 ilustra uma *query* que especifica uma janela temporal.

```
select * from Withdrawal.win:time(4 sec)
```

Figura 18. Exemplo de *query* utilizando janela temporal

Essa janela temporal, *win:time(4 sec)*, mantém armazenada uma quantidade ilimitada de eventos que foram recebidos nos últimos 4 segundos. A Figura 19 ilustra um diagrama contendo o mesmo fluxo de eventos apresentado na Figura 17, porém, considerando a *query* especificada na Figura 18.

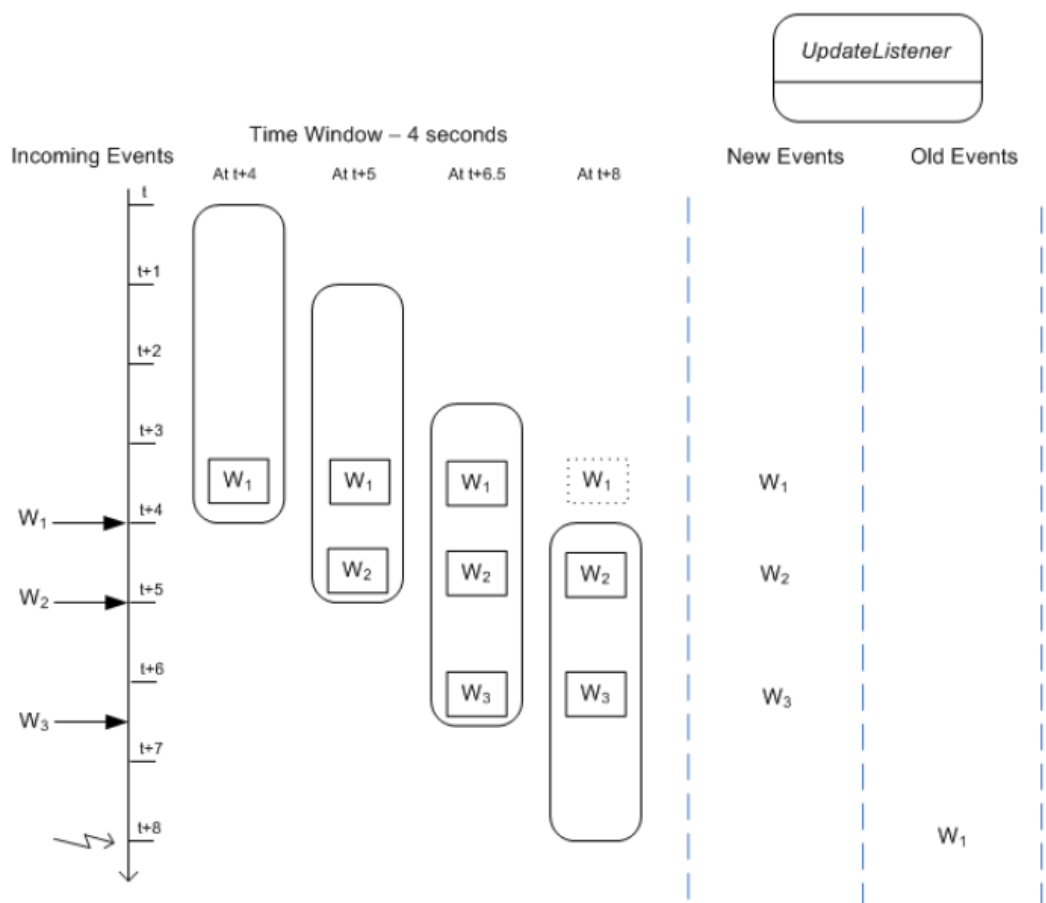


Figura 19. Modelo de processamento utilizando janela temporal

A cada novo segundo, a janela é deslocada ao longo do tempo, armazenando todos os eventos contidos nos últimos 4 segundos. Quando o primeiro evento (W_1) é recebido, no período de tempo $t+4$, o mesmo é enviado aos métodos *update* dos *Listeners* correspondentes como um novo evento. No tempo $t+8$, como o limite de tempo de 4 segundos da janela foi atingido para o primeiro evento (W_1) ele é

enviado aos métodos *update* dos *Listeners* correspondentes como um evento antigo, independentemente do número de eventos contidos na janela ou da chegada de um novo evento.

A Figura 20 ilustra uma *query* utilizando janela de tamanho 5, juntamente com a utilização de um filtro, restringindo o valor do atributo *amount* para valores maiores ou iguais a 200. O modelo de processamento correspondente é ilustrado pelo diagrama da Figura 21.

```
select * from Withdrawal(amount>=200).win:length(5)
```

Figura 20. Exemplo de *query* utilizando filtros

Nesse modelo de processamento, eventos continuam sendo enviados aos *Listeners* como um novo evento ao entrar na janela e, analogamente, como um evento antigo ao sair da janela. A diferença é dada pela utilização do filtro, permitindo que apenas eventos que satisfaçam a condição estabelecida sejam adicionados à janela. Filtros são representados por restrições aplicadas sobre a cláusula *from* por meio de parênteses. Considerando um fluxo de eventos com os mesmos seis eventos anteriores do tipo *Withdrawal*, apenas três eventos (W1, W3 e W6) foram enviados como um novo evento, enquanto nenhum evento foi enviado como um evento antigo.

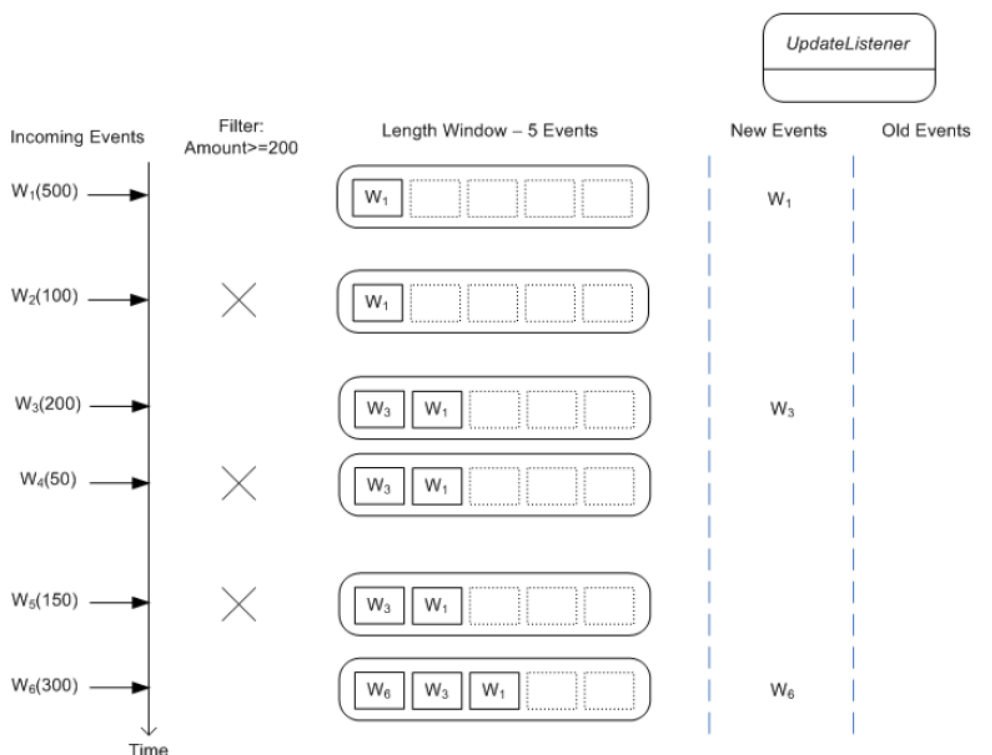


Figura 21. Modelo de processamento utilizando filtros

A Figura 22 apresenta uma *query* que aparenta ser semanticamente equivalente à apresentada pela Figura 20. A mesma restrição é aplicada sobre o atributo *amount*, porém utilizando a cláusula *where* ao invés de filtros. Apesar de ambas utilizem a mesma restrição, o resultado difere se considerarmos o modelo de processamento utilizando filtros e a calcula *where*. O modelo de processamento correspondente à *query* apresentada na Figura 22 é ilustrado na Figura 23.

```
select * from Withdrawal.win:length(5) where amount >= 200
```

Figura 22. Exemplo de *query* utilizando a cláusula *where*

Ao contrário do que ocorre na utilização de filtros, ao utilizar a cláusula *where* todos os eventos são inseridos na janela, independentemente das restrições aplicadas sobre os mesmos serem avaliadas como verdadeira ou falsa. A restrição é avaliada apenas após a inserção na janela. Caso a restrição seja satisfeita, o evento correspondente é enviado ao *Listener* como um novo evento. Caso contrário, o evento não é enviado, porém, continua armazenado na janela. Isso provoca o envio

do primeiro evento (W1) como um evento antigo durante a chegada do sexto evento (W6), diferentemente do que ocorreu durante a utilização do filtro.

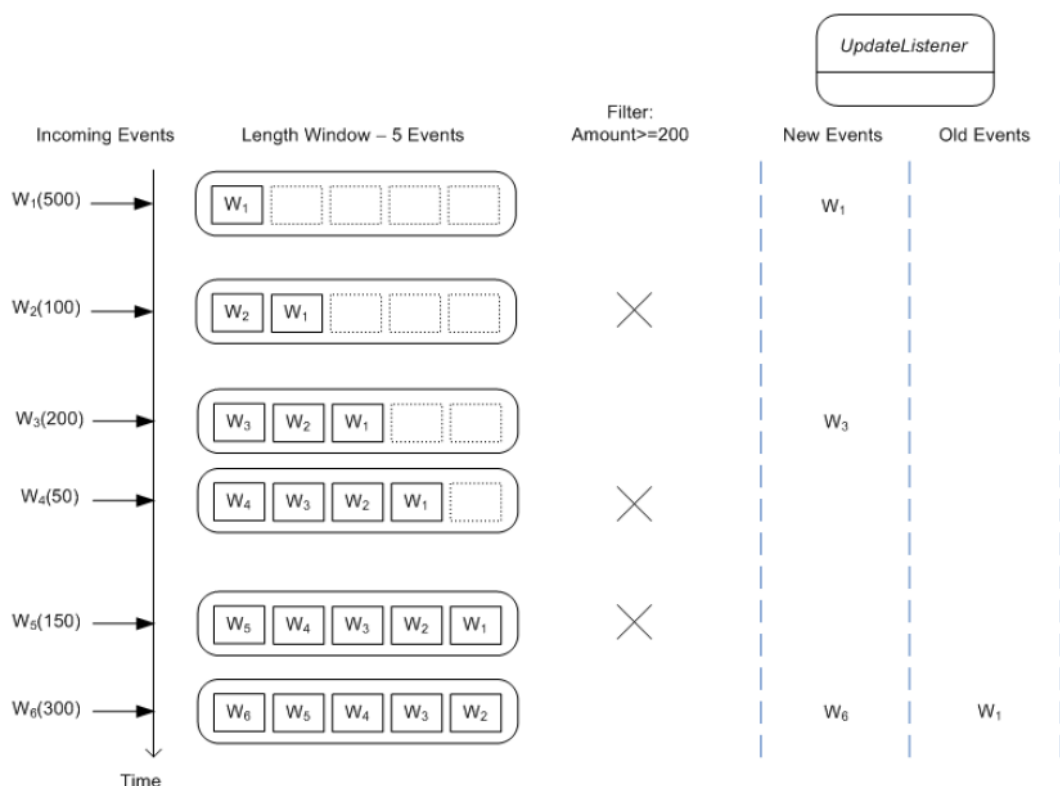


Figura 23. Modelo de processamento utilizando a cláusula *where*

3.4 EVENT PATTERNS

Esper provê construções em sua linguagem EPL que permitem a especificação de uma expressão baseada no *matching* (casamento) de padrões de eventos (*event patterns*). Quando um ou mais eventos satisfazem as condições definidas em um padrão de eventos, ocorre o *matching* (ou casamento) entre o padrão de eventos e os eventos detectados pelo Esper. Padrões de eventos são implementados em Esper através da utilização de máquinas de estados, permitindo o processamento de sequências ou combinação de eventos (ou a não ocorrência de eventos). Além disto, pode-se relacionar os eventos por meio de seus atributos e operadores temporais.

Padrões são definidos a partir de dois tipos de construções: (i) estruturas atômicas e (ii) operadores. Estruturas atômicas consistem nos blocos básicos

utilizados na construção de padrões, como filtros e observadores. Operadores são utilizados para controlar o ciclo de vida das expressões e combinar as estruturas atômicas de forma lógica ou temporal. Um padrão pode ser utilizado em qualquer lugar da cláusula *from* de uma *query* EPL, através do operador *pattern*. Padrões podem ser usados em combinação com a cláusula *where*, *group by* e *having*, além de ser possível especificar janelas sobre os mesmos. Dentre os operadores, os mais utilizados e importantes para a elaboração dos padrões propostos pelo presente trabalho consistem nos operadores *followed by* e *every*. As seções seguintes discutem esses operadores em detalhes.

3.4.1 Operador Followed By

O operador *followed by*, representado por uma seta direcional “->”, é aplicado sobre duas (sub) expressões, uma do lado esquerdo e outra do lado direito. Esse operador especifica que, primeiramente, a expressão do lado esquerdo deve ser avaliada como verdadeira e, somente após isso, a expressão do lado direito é avaliada. Assim que ambas as subexpressões forem avaliadas como verdadeiras, a expressão inteira é avaliada como verdadeira. A Figura 24 ilustra dois padrões para exemplificar a utilização do operador *followed by*. O primeiro padrão especifica a ocorrência de um evento do tipo B, seguida da ocorrência de um evento do tipo A. O operador *followed by* inicia a verificação da ocorrência de um evento do tipo B (e possíveis restrições associadas ao mesmo) somente após detectar a ocorrência de um evento do tipo A. Quaisquer eventos do tipo B recebidos antes da ocorrência de um evento do tipo A são ignorados pelo padrão em questão.

Em Esper, padrões são avaliados uma única vez. Assim, o primeiro padrão, ilustrado pela Figura 24, tem como resultado do *matching* apenas a primeira tupla formada por um evento do tipo A e um evento do tipo B (assumindo que o evento do tipo B tenha sido recebido posteriormente ao evento do tipo A). Para que um padrão seja avaliado mais de uma vez, tendo como resultado do *matching*, por exemplo, todas as tuplas formadas por eventos do tipo A e eventos do tipo B, deve ser utilizado o operador *every* (seção 3.4.2)

O segundo padrão apresenta dois operadores novos: *timer:interval* e *not*. O operador *timer:interval* aguarda um tempo igual ao tempo especificado como parâmetro. Passado esse tempo, o operador *timer:interval* retorna *true*. Seu papel pode ser comparado ao de um contador. Já o operador *not* especifica a negação no valor de uma expressão. Ao ser aplicado sobre um evento, esse operador especifica o interesse na não ocorrência do mesmo. Como Esper trabalha com fluxos de eventos contínuos e infinitos, o operador *not* separadamente, aplicado sobre um evento, não tem significado real, uma vez que é apenas possível inferir a não ocorrência de um evento quando todos os eventos já foram recebidos, fato este que nunca ocorrerá. Por este motivo, o operador *not* deve sempre estar associado a algum intervalo. O intervalo em questão é especificado pelo operador *timer:interval* (no exemplo, delimitando um intervalo de 5 min). Assim, segundo padrão tem como objetivo detectar a ocorrência de um evento do tipo A no qual não ocorra a detecção de algum evento do tipo B no período de cinco minutos após sua detecção. O resultado do *matching* corresponde a uma tupla contendo apenas um evento do tipo A.

```
(1) A -> B  
(2) A -> (timer:interval(5 min) and not B)
```

Figura 24. Exemplo do uso do operador *followed by*

3.4.2 Operador Every

O operador *every* indica que uma expressão (ou subexpressão) de um padrão deve ser reiniciada quando for avaliada como verdadeira ou falsa. Conforme discutido na seção 3.4.1, padrões são avaliados uma única vez. Sem o uso do operador *every*, uma vez que uma subexpressão seja avaliada como verdadeira ou falsa, ela deixará de ser avaliada permanentemente após a primeira ocorrência. O operador *every* pode ser entendido como uma *factory* de subexpressões: quando é

avaliada como verdadeira, o operador *every* inicia uma nova subexpressão, ficando à espera de novos eventos que satisfaçam a nova subexpressão.

Tomemos como exemplo os padrões especificados pela Figura 25. O primeiro padrão apenas aguarda pela primeira ocorrência de um evento do tipo A. Uma vez que seja detectado um evento do tipo A, esse padrão deixa de ser avaliado, permanentemente. Já no segundo padrão, uma vez que um evento do tipo A seja detectado, uma nova subexpressão é criada, aguardando por um novo evento do tipo A. Assim, esse padrão será ativado para todos os eventos do tipo A recebidos ao longo do tempo, e não apenas uma vez, como ocorre no primeiro padrão.

(1)	A
(2)	every A

Figura 25. Exemplo do uso do operador *every*

3.4.3 Combinando os operadores *every* e *followed by*

É possível combinar os dois operadores apresentados anteriormente, *followed by* e *every*. A Figura 26 ilustra o exemplo de um fluxo de eventos, contendo eventos dos tipos A, B, C, D, E e F, juntamente com quadro padrões diferentes utilizando o operador *every* em combinação com o operador *followed by*.

A ₁	B ₁	C ₁	B ₂	A ₂	D ₁	A ₃	B ₃	E ₁	A ₄	F ₁	B ₄
(1) every (A -> B)						(2) every A -> B					
(3) A -> every B						(4) every A -> every B					

Figura 26. Exemplo do uso do operador *every* combinado com o operador *followed by*

No primeiro padrão, o operador *every* é aplicado sobre a subexpressão “A->B”, ou seja, um evento do tipo A seguido de um evento do tipo B. Inicialmente, o padrão

aguarda por um evento do tipo A, ignorando todos os outros eventos recebidos (inclusive eventos do tipo B). Uma vez detectado um evento do tipo A, novos eventos do tipo A são ignorados pelo padrão, até que seja encontrado algum evento do tipo B. Quando um evento do tipo B é detectado, o padrão todo é avaliado como verdadeiro. Nesse momento, o operador *every* reinicia a subexpressão, aguardando novamente por novos eventos do tipo A. Considerando o fluxo de eventos da Figura 26, as seguintes tuplas levariam a avaliação do primeiro como verdadeiro: {A₁, B₁}, {A₂, B₃} e {A₄, B₄}.

Por sua vez, no segundo padrão, temos o operador *every* aplicado apenas à subexpressão “A”. Essa subexpressão então é relacionada à subexpressão “B”, por meio do operador *followed by*. Nesse caso, sempre que um evento do tipo A é detectado, uma nova subexpressão é criada, aguardando um novo evento do tipo A. Assim, duas subexpressões encontram-se ativas durante um mesmo instante no tempo, uma subexpressão aguardando um evento do tipo B, e a nova subexpressão, aguardando um evento do tipo A. À medida que um evento do tipo B seja detectado, a primeira subexpressão é encerrada, restando apenas a segunda subexpressão. Uma nova subexpressão é criada apenas quando a subexpressão, na qual o operador *every* é aplicado, é avaliada como verdadeira, ou seja, quando um evento do tipo A é detectado. Considerando o fluxo de eventos da Figura 26, as seguintes tuplas levariam a avaliação do segundo padrão como verdadeiro: {A₁, B₁}, {A₂, B₃}, {A₃, B₃} e {A₄, B₄}.

O terceiro padrão aplica o operador *every* sobre a subexpressão “B”. Essa subexpressão, por sua vez, corresponde à subexpressão do lado direito do operador *followed by*, enquanto a subexpressão “A” corresponde à subexpressão do lado esquerdo. A primeira subexpressão, como não possui o operador *every*, é avaliada apenas uma vez, ou seja, aguardando pela primeira ocorrência de um evento do tipo A. A partir desse momento, sempre que um evento do tipo B é detectado, uma nova subexpressão é criada, aguardando por um novo evento do tipo B. Isso faz com que este padrão seja ativado para todas as combinações entre algum evento do tipo B e o primeiro evento do tipo A detectado, levando em consideração a restrição do operador *followed by* (i.e., o evento do tipo B deve ser detectado após o evento do tipo A). Considerando o fluxo de eventos da Figura 26, as seguintes tuplas levariam

a avaliação do primeiro padrão como verdadeiro: $\{A_1, B_1\}$, $\{A_1, B_2\}$, $\{A_1, B_3\}$ e $\{A_1, B_4\}$.

Por fim, o quarto padrão aplica o operador *every* duas vezes, sobre as subexpressões “A” e “B”, ambas relacionadas pelo operador *followed by*. Esse padrão foi amplamente utilizado na construção dos padrões propostos pelo presente trabalho, discutidos no capítulo 5. Quando um evento do tipo A é detectado, uma nova subexpressão é criada, aguardando por novos eventos do tipo A. A subexpressão contendo o evento do tipo A funciona de forma análoga ao terceiro padrão: cada novo evento do tipo B torna o padrão verdadeiro e, ao mesmo tempo, cria uma nova subexpressão, aguardando novos eventos do tipo B. Dessa forma, o quarto padrão é ativado para todas as combinações entre algum evento do tipo A e um evento do tipo B, levando em consideração a restrição do operador *followed by*. Considerando o fluxo de eventos da Figura 26, as seguintes tuplas levariam a avaliação do primeiro padrão como verdadeiro: $\{A_1, B_1\}$, $\{A_1, B_2\}$, $\{A_1, B_3\}$, $\{A_1, B_4\}$, $\{A_2, B_3\}$, $\{A_2, B_4\}$, $\{A_3, B_3\}$, $\{A_3, B_4\}$ e $\{A_4, B_4\}$.

3.5 EXEMPLO DE APLICAÇÃO UTILIZANDO A PLATAFORMA ESPEER

Esta seção tem como objetivo apresentar os passos necessários para a criação de uma aplicação utilizando a plataforma Esper⁴. É assumido que o usuário possui a plataforma corretamente configurada em seu ambiente de trabalho, bem como suas respectivas dependências devidamente resolvidas. Tal explicação foge do escopo do presente trabalho.

As instruções e exemplos aqui apresentados tem como base o *quick start guide*⁵ disponibilizado pela plataforma Esper. O primeiro passo consiste na implementação dos arquivos que representam, programaticamente, os eventos do domínio. Conforme discutido anteriormente, em Esper, eventos podem ser representados como classes Java, documentos XML ou Map. A forma mais comum, e utilizada neste exemplo, consiste em classes Java segundo o padrão POJO. Por

⁴ <http://www.espertech.com/esper/download.php>

⁵ <http://www.espertech.com/esper/quickstart.php>

se tratar de um exemplo simples, vamos considerar apenas um tipo de evento, ilustrado pela Figura 27

```
1 public class OrderEvent {  
2     private String itemName;  
3     private double price;  
4  
5     public OrderEvent(String itemName, double price) {  
6         this.itemName = itemName;  
7         this.price = price;  
8     }  
9  
10    public String getItemName() {  
11        return itemName;  
12    }  
13  
14    public double getPrice() {  
15        return price;  
16    }  
17 }
```

Figura 27. Classe Java correspondente ao tipo de evento *OrderEvent*.

Esse tipo de evento, denominado *OrderEvent*, possui apenas dois atributos, *itemName* e *price* (linhas 2 e 3, Figura 27). Além disso, é possível observar o método construtor desse tipo de evento, bem como seus respectivos métodos *get* de cada atributo. Caso o método construtor não fosse implementado, seria necessário implementar também os respectivos métodos *set* para cada atributo. Definidos os tipos de eventos do domínio, é possível criar *queries* sobre fluxos de eventos desses tipos. A Figura 28 ilustra uma *query* baseada no tipo de evento *OrderEvent*.

```
select avg(price) from org.myapp.event.OrderEvent.win:time(30 sec)
```

Figura 28. Exemplo de *query* utilizando o tipo de evento *OrderEvent*

Essa *query* tem como objetivo calcular a média (*avg*) do campo *price* de todos os eventos do tipo *OrderEvent* nos últimos 30 segundos (*win:time(30 sec)*). Observe que na cláusula *from* é utilizado o nome qualificado do tipo de evento, assumindo que o mesmo esteja no pacote *org.myapp.event*. Para utilizar apenas o nome da

classe, os nomes qualificados dos pacotes devem ser especificados à plataforma Esper por meio de configurações específicas.

Uma vez definida a *query*, é necessário implementar uma classe que represente o *Listener* invocado quando as condições definidas na *query* forem satisfeitas. Essa classe deve implementar a classe *UpdateListener*, implementando o método *update* correspondente. Esse Listener simples, denominado *MyListener*, apenas exibe a média registrada pela *query* em questão (a média dos valores *price* dos eventos do tipo *OrderEvent* nos últimos 30 segundos). A Figura 29 apresenta a implementação da classe *MyListener*.

```
1 public class MyListener implements UpdateListener {  
2     public void update(EventBean[] newEvents, EventBean[] oldEvents) {  
3         EventBean event = newEvents[0];  
4         System.out.println("avg=" + event.get("avg(price)"));  
5     }  
6 }
```

Figura 29. Exemplo de *Listener*

Definidos os tipos de eventos, as *queries* e os *Listeners*, é possível iniciar a plataforma Esper para monitorar tal *query*. A Figura 30 ilustra o exemplo de uma aplicação utilizando a plataforma Esper. Inicialmente, deve se armazenar uma instância da máquina Esper, através do método *getDefaultProvider*, disponível na classe *EPServiceProviderManager* (linha 1, Figura 30). Feito isso, é criada uma variável do tipo *String* contendo a *query* que deseja-se monitorar (linha 2, Figura 30). Esse *query* é registrada na máquina Esper através do método *createEPL* (linha 3, Figura 30), por meio da interface *EPAdministrator*, responsável pelo gerenciamento de todas as *queries* na máquina Esper. Para ter acesso ao *EPAdministrator*, o mesmo deve ser requisitado à máquina Esper por meio do método *getEPAdministrator*.


```

1 EPServiceProvider epService = EPServiceProviderManager.getDefaultProvider();
2 String expression = "select avg(price) from org.myapp.event.OrderEvent.win:time(30 sec)";
3 EPStatement statement = epService.getEPAdministrator().createEPL(expression);
4
5 MyListener listener = new MyListener();
6 statement.addListener(listener);
7
8 OrderEvent event = new OrderEvent("shirt", 74.50);
9 epService.getEPRuntime().sendEvent(event);

```

Figura 30. Exemplo de aplicação utilizando a plataforma Esper

Queries registradas na máquina Esper devem ser associada a um *Listener* (linhas 5 e 6, Figura 30), de forma que seja possível reagir à ocorrência das mesmas. Assim que as condições estabelecidas numa *query* são satisfeitas, é invocado o método *update* do *Listener* correspondente. Para a *query* apresentada na Figura 28 não foi especificada nenhum tipo de restrição, de forma que cada novo evento do tipo *OrderEvent* leva à invocação do método *update*. Esse método apenas imprime no dispositivo de saída padrão a média do campo *price* calculada a partir dos eventos desse tipo recebido nos últimos 30 segundos (linhas 2 a 4, Figura 29).

Por fim, as linhas 8 e 9 da Figura 30 ilustram a criação de um evento do tipo *OrderEvent* e seu envio à máquina Esper. O envio é realizado através do método *sendEvent*, disponibilizado pela interface *EPRuntime*. Essa interface é responsável por enviar eventos à máquina Esper, bem como alterar o valor de variáveis dos mesmos ou executar *queries* sobre demanda (caso seja necessário avaliar *queries* em momentos específicos e não de forma constante). Para ter acesso ao *EPRuntime*, o mesmo deve ser requisitado à máquina Esper por meio do método *getEPRuntime*.

3.6 CONSIDERAÇÕES DO CAPÍTULO

Este capítulo apresentou aspectos importantes da plataforma Esper que são utilizados no desenvolvimento deste trabalho. Foram discutidos aspectos gerais da plataforma, cujo modelo de processamento de eventos utilizado foi apresentado e discutido. Foram apresentados alguns motivos nos quais levaram a utilização de tal plataforma neste trabalho, dentre o qual se destaca sua *event processing language*, denominada Esper EPL, no qual mostrou-se uma linguagem expressiva, permitindo

que sejam definidos padrões complexos avaliados sobre diferentes fluxos de eventos. Essa linguagem foi introduzida, com a utilização de exemplos, bem como foram apresentados e discutidos seus principais operadores utilizados pelo presente trabalho na construção dos padrões apresentados no capítulo 5.

O capítulo seguinte apresenta a plataforma SIMPLE, cujo seu desenvolvimento utilizou como base a plataforma Esper.

4 SIMPLE: SITUATION MAPPING LAYER FOR ESPEr

Conforme discutido na seção 1.1, tradicionalmente, sistemas de CEP não oferecem suporte para o gerenciamento do ciclo de vida das situações, segundo a definição de situação adotada pelo presente trabalho. A fim de permitir a detecção de processamento de situações na plataforma Esper, uma das contribuições deste trabalho consiste no desenvolvimento da plataforma SIMPLE (*Situation Mapping Layer for Esper*)⁶, de acordo com o objetivo 2 (seção 1.2).

Este capítulo está organizado da seguinte forma: a Seção 4.1 apresenta o Modelo de Detecção de Situações em CEP, desenvolvido e utilizado pelo presente trabalho como base para a criação da plataforma SIMPLE; a Seção 4.2 apresenta as informações necessárias para representar e definir um tipo de situação na plataforma SIMPLE; a Seção 4.3 apresenta os conceitos de composicionalidade e integridade das situações e como são tratados na plataforma SIMPLE; a Seção 4.4 apresenta a visão do usuário sobre a arquitetura e informações de utilização da plataforma SIMPLE; a Seção 4.5 apresenta a visão interna da plataforma SIMPLE, discutindo detalhes a respeito dos componentes internos da plataforma; a Seção 4.6 apresenta as etapas necessárias para a criação de uma instância de um tipo de situação, representando sua respectiva ativação, utilizando a plataforma SIMPLE e, por fim, a Seção 4.7 apresenta as considerações do capítulo.

4.1 MODELO DE DETECÇÃO DE SITUAÇÕES EM CEP

Uma das contribuições deste trabalho consiste no desenvolvimento do Modelo de Detecção de Situações em CEP. A elaboração deste modelo possibilitou um melhor entendimento a respeito da utilização de situações em sistemas de CEP. O principal ponto deste modelo consistiu em esclarecer as relações existentes entre situações e eventos, relações essas que influenciaram a implementação do conceito de situação (juntamente com seus respectivos ciclos de vida) na plataforma SIMPLE. Tais relações e escolhas influenciadas por este modelo são discutidas ao

⁶ O código fonte completo encontra-se disponível em: <https://github.com/Simonassi/SIMPLE>

como “uma pessoa com temperatura corporal maior que 36 graus Celsius”. Um possível evento do tipo *Situation Creation Trigger Event* é dado por um evento contendo a informação “temperatura de João igual a 38 graus celsius”. Toda instância de um tipo de situação deve estar associada a um evento do tipo *Situation Creation Trigger Event*, e somente um.

De forma análoga, um evento do tipo *Situation Deactivation Trigger Event* representa o evento (ou, novamente, o conjunto de eventos) que levou à desativação da situação. Para o exemplo anterior, um possível evento do tipo *Situation Deactivation Trigger Event* é dado por um evento contendo a informação “temperatura de João igual a 35 graus celsius”. Uma situação pode ficar ativada por tempo indeterminado, e pode nunca ser desativada. Dessa forma, uma instância de um tipo de situação pode estar associada a um ou nenhum evento do tipo *Situation Creation Trigger Event*.

Um evento do tipo *Situation Creation Event* representa o evento criado pela plataforma CEP ao detectar a ativação de uma situação (*i.e.*, ao detectar a ocorrência de um evento do tipo *Situation Creation Trigger Event*). Para o exemplo da situação Febre, um possível evento do tipo *Situation Creation Event* é dado por um evento contendo a informação “situação Febre de João ativada”. Esse evento é sempre um evento simples. Assim como ocorre com o *Situation Creation Trigger Event*, toda instância de um tipo de situação deve estar associada a um evento do tipo *Situation Creation Event*, e somente um.

De forma análoga, um evento do tipo *Situation Deactivation Event* representa o evento criado pela plataforma CEP ao detectar a desativação de uma situação (*i.e.*, detectar a ocorrência de um evento do tipo *Situation Deactivation Trigger Event*). Novamente utilizando o exemplo da situação Febre, um possível evento do tipo *Situation Deactivation Event* é dado por um evento contendo a informação “situação Febre de João desativada”. Esse evento sempre é um evento simples. Assim como ocorre com o *Situation Creation Trigger Event*, toda instância de um tipo de situação pode estar associada a um ou nenhum evento do tipo *Situation Deactivation Event*.

Os quatro eventos citados acima (*Situation Creation Trigger Event*, *Situation Deactivation Trigger Event*, *Situation Creation Event* e *Situation Deactivation Event*),

possibilitaram um entendimento a respeito da definição de uma situação (*Situation*) em sistemas baseados em eventos, esclarecendo a relação entre o conceito de situação e eventos em CEP. Esses eventos revelaram a necessidade de definir duas regras para cada tipo de situação: uma regra de ativação e uma regra de desativação.

Uma regra de ativação corresponde a um padrão que define um *Situation Creation Trigger Event* e, uma vez que tal padrão seja satisfeito, ele é responsável por criar um *Situation Creation Event* correspondente. De forma análoga, uma regra de desativação corresponde a um padrão que define um *Situation Deactivation Trigger Event* e, uma vez que tal padrão seja satisfeito, ele é responsável por criar um *Situation Deactivation Event* correspondente. Os padrões definidos e utilizados pelo presente trabalho para representar as regras de ativação e desativação dos tipos de situações são apresentados em detalhes no capítulo 5. Uma vez que o presente trabalho utiliza a plataforma Esper como base, esses padrões correspondem à *queries EPL*.

Os eventos do tipo *Situation Creation Event* e *Situation Deactivation Event* representam eventos criados pela plataforma SIMPLE e enviados aos respectivos clientes, por meio da plataforma Esper, de forma que sejam notificados sobre a ativação ou desativação dos tipos de situação de interesse. As seções seguintes detalham como a plataforma SIMPLE manipula (cria, envia e processa) tais eventos. *Situation Creation Events* e *Situation Deactivation Events* indicam qual a última alteração no *status* de uma instância de um tipo de situação: se o último evento gerado referente a uma instância é do tipo *Situation Creation Event*, essa instância encontra-se ativa. Caso o último evento gerado referente a uma instância é do tipo *Situation Deactivation Event*, essa instância encontra-se desativada. Essas informações permitem que a plataforma SIMPLE gerencie o ciclo de vida das situações.

4.2 SITUAÇÕES EM SIMPLE

Essa seção descreve, de forma geral, o processo de definição de um tipo de situação utilizando a plataforma SIMPLE. Um tipo de situação é representado por uma classe Java definida segundo o padrão POJO (*Plain Old Java Objects*) (Richardson, 2006). A criação da classe Java correspondente deve ser realizada utilizando o componente *Situation*: uma classe abstrata, disponibilizada pela plataforma SIMPLE, responsável por definir os atributos e métodos comuns aos tipos de situações. Toda classe que representa um tipo de situação devem estender esta classe. A Figura 35 ilustra, de forma simplificada, a implementação do componente *Situation*.

```

1 public abstract class Situation {
2     private boolean activated;
3     private Date start_time;
4     private Date end_time;
5     private String sitName, eplA, eplD;
6     private int id;
7     private String key;
8
9     public abstract Object doActionAtCreateDeactivationEvent();
10    public abstract Situation createNewSit(EventBean event);
11    public Object doActionAtCreateActivationEvent(EventBean[] newEvents){ return null; }
12
13    public void doActionAtDeactivation(EventBean event){
14        System.out.println(
15            "Situation " + getSitName()
16            + " deactivated. ID = " + getId()
17            + " Start Time = " + getStart_time()
18            + "End Time = " + getEnd_time()
19        );
20    }
21
22    public void doActionAtActivation(EventBean event){
23        System.out.println(
24            "Situation " + getSitName()
25            + " activated. ID = " + getId()
26            + " Start Time = " + getStart_time()
27            + "End Time = " + getEnd_time()
28        );
29    }
30 }

```

Figura 32. Componente *Situation*

O atributo *activated* (linha 2, Figura 35) é responsável por indicar o *status* das instâncias do tipo de situação correspondente: ativada ou desativada. Uma instância de um tipo de situação é dita ativada caso seu atributo *activated* possua valor igual à *true*. Da mesma forma, uma instância de um tipo de situação é dita desativada caso

seu atributo *activated* possua valor igual à *false*. Os atributos *start_time* e *end_time* (linhas 3 e 4, Figura 35) indicam, respectivamente, os tempos de ativação e desativação de cada instância do tipo de situação correspondente. Instâncias ativadas possuem o valor do atributo *end_time* igual à *null*. O atributo *id* (linhas 6, Figura 35) é utilizado para controle interno da plataforma SIMPLE, identificando unicamente cada instância de um tipo de situação, de forma a garantir a integridade (seção 4.3) das situações. Sua atribuição deve ser realizada exclusivamente pela plataforma SIMPLE.

O atributo *sitName* (linha 5, Figura 35) contém o nome do tipo de situação correspondente, associando a classe Java ao seu respectivo tipo de situação. Por sua vez, os atributos *epIA* e *epID* (linha 5, Figura 35) são responsáveis por armazenar, respectivamente, as regras de ativação e desativação de cada tipo de situação, especificadas através da linguagem Esper EPL (capítulo 3), utilizando como base os padrões apresentados e discutidos no capítulo 5.

Assumindo o modelo de situação ilustrado pela Figura 9, a implementação da classe que representa o tipo de situação *Fever* é ilustrada nas figuras Figura 33 e Figura 34. Por simplicidade, os métodos *get* e *set* de cada atributo foram omitidos.

Na Figura 33 é possível observar o atributo *patient*, responsável por armazenar uma referência ao Paciente participante da situação *Fever*. Toda entidade dita participante de um tipo situação, deve possuir um atributo correspondente na respectiva classe Java que o representa. O método construtor de uma classe que representa um tipo de situação (linhas 4 a 26, Figura 33) deve, obrigatoriamente, inicializar o nome da situação juntamente com suas regras de ativação e desativação (atributos *sitName*, *epIA* e *epID*), por meio dos métodos *setSitName*, *setEpIA* e *setEpID*, respectivamente (linhas 6, 8 e 15, Figura 33).


```

1 public class Fever extends Situation {
2     private Patient patient;
3
4     public Fever(){
5
6         setSitName("Fever");
7
8         setEplA("select "
9             + "    patient, Patient.key as key1 "
10            + "from "
11            + "    Patient as patient "
12            + "where "
13            + "    temperature > 36 ");
14
15        setEplD("select Patient.key as key1 "
16            + "from "
17            + "    Fever.std:unique(id) as Fever, "
18            + "    Patient.std:unique(key) as patient "
19            + "where "
20            + "    Fever.activated = true "
21            + "    and"
22            + "    ("
23            + "        Fever.patient.key = patient.key and "
24            + "        not(patient.temperature > 36) "
25            + "    )" );
26    }

```

Figura 33. Primeira parte da implementação da classe *Fever*

Na segunda parte da implementação da classe *Fever*, ilustrada pela Figura 34, é possível observar dois métodos: *createNewSit* (linhas 28 a 33, Figura 34) e *doActionAtCreateDeactivationEvent* (linhas 35 a 40, Figura 34). Esses métodos abstratos, contidos na classe *Situation*, devem ser implementados pelo usuário. Eles representam, respectivamente, métodos construtores dos eventos de ativação e desativação do tipo de situação correspondente, especificando a estrutura interna de cada um desses eventos.

Os métodos *createNewSit* e *doActionAtCreateDeactivationEvent* são análogos e seguem um padrão em suas implementações. No método *createNewSit*, inicialmente, é criado um objeto do tipo de situação em questão, representando o evento de ativação (linha 29, Figura 34). Cada atributo do objeto criado deve ter seu valor atribuído através do objeto *event*, utilizado como parâmetro da função *createNewSit*. Isso é realizado através do método *event.get(String)*. O argumento do tipo *String* passado a esse método deve corresponder ao nome utilizado como apelido da entidade deste mesmo tipo (através do operador *as*, definido na cláusula

from da respectiva regra de ativação (linha 11, Figura 33)). Nesse caso, o valor corresponde é a String “*patient*”. A atribuição ao atributo do tipo Paciente é ilustrada na linha 30 da Figura 34. Por fim, o objeto criado é retornado (linha 32, Figura 34).

```
27
28     public Situation createNewSit(EventBean event) {
29         Fever fever = new Fever();
30         fever.setPatient((Patient)event.get("Patient"));
31
32         return fever;
33     }
34
35     public Object doActionAtCreateDeactivationEvent() {
36         Fever fever = new Fever();
37         fever.setPatient(this.getPatient());
38
39         return fever;
40     }
41
42 }
```

Figura 34. Segunda parte da implementação da classe *Fever*

A implementação do método *doActionAtCreateDeactivationEvent* é análoga, conforme dito anteriormente. A única diferença consiste na ausência do parâmetro *event*. Dessa forma, ao atribuir os valores dos atributos referentes ao objeto criado, basta utilizar a palavra reservada *this*, ao invés do método *event.get* (linha 41, Figura 34).

Classes que estendem o tipo *Situation* possuem dois métodos cujas implementações são opcionais: *doActionAtAtivation* e *doActionAtDeactivation*. Esses métodos representam, respectivamente, as ações a serem executadas pela aplicação cliente durante a ativação e desativação do tipo de situação correspondente. Por padrão, esses métodos apenas listam o nome do tipo de situação, um identificador único para cada instância e seus tempos de ativação e desativação. Caso o cliente necessite executar ações mais complexas, por exemplo, enviar uma mensagem de alerta para o administrador do sistema, é possível sobrecarregar tais métodos.

4.3 INTEGRIDADE E COMPOSICIONALIDADE DE SITUAÇÕES

A plataforma SIMPLE foi desenvolvida com base na plataforma Esper, de modo a auxiliar na especificação e detecção de processamento de situações, levando em consideração o modelo de eventos apresentado na seção 4.1. SIMPLE gerencia o ciclo de vida das situações, promovendo a composicionalidade e integridade das situações, como explicado a seguir.

Composicionalidade diz respeito ao fato de que tipos de situações podem ser especificados a partir da composição de tipos de situações definidos previamente. Tomemos como exemplo um tipo de situação Febre definido como “uma pessoa com temperatura corporal maior que 37 graus célsius”. A partir desse tipo de situação, é possível especificar um tipo de situação Febre Intermitente definido como “uma pessoa que participou de duas ocorrências de situações do tipo Febre no último mês”. Ao especificar o tipo de situação Febre Intermitente as restrições contidas no tipo de situação Febre devem ser transparentes ao usuário, uma vez que já foram definidas previamente no tipo de situação correspondente.

Integridade diz respeito à manutenção do estado correto das situações (*i.e.*, ativada ou desativada) e à identificação única de cada instância de um tipo de situação. Ao utilizar sistemas baseados em eventos, uma vez que um padrão é avaliado como verdadeiro (ou parcialmente verdadeiro), o mesmo não é reavaliado. Tomemos como exemplo um tipo de situação *SituationC*, definido a partir da composição de dois outros tipos de situação, *SituationA* e *SituationB*. A Figura 35 ilustra a regra de ativação correspondente ao tipo de situação *SituationC*, especificada utilizando a linguagem Esper EPL (seção 3.4). Essa regra especifica que a situação deve permanecer ativa se, e somente se, duas instâncias dos tipos de situação *SituationA* e *SituationB* encontram-se ativas (*i.e.*, o atributo *activated* possui valor igual à *true*) em um mesmo instante do tempo.

```
1 select *  
2 from SituationA(activated = true),  
3     SituationB(activated = true)
```

Figura 35. Regra de ativação referente ao tipo de situação *SituationC*

A Figura 36 apresenta um possível fluxo de eventos para esse exemplo. Nesse fluxo é possível observar dois eventos, A_1 e A_2 , representando eventos de ativação referentes a duas instâncias distintas do tipo de situação *SituationA*, juntamente com dois eventos A_{1D} e A_{2D} , representando eventos de desativação referentes aos eventos de ativação do tipo de situação *SituationA*, A_1 e A_2 , respectivamente. Além disso, é possível observar um evento B_1 , representando um evento de ativação referente a uma instância do tipo de situação *SituationB*.

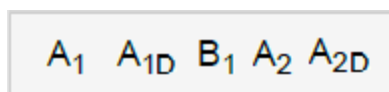


Figura 36. Exemplo de fluxo de dados de eventos do tipo *SituationA* e *SituationB*

Uma vez que um evento A_1 é recebido pelo Esper ocorre uma avaliação parcial na regra (padrão) de ativação descrita pela Figura 35. A primeira condição especificada faz-se verdadeira, ou seja, um evento de ativação do tipo *SituationA*. Essa condição, uma vez satisfeita, não é mais avaliada. Apenas a segunda condição continuará a ser avaliada, aguardando um evento de ativação do tipo *SituationB*. Em seguida, é recebido um evento A_{1D} , indicando que a instância da situação do tipo *SituationA*, ativada pelo evento A_1 , encontra-se agora, desativada. Como a condição referente ao tipo de situação *SituationA* foi previamente avaliada como verdadeira, essa informação é descartada. Assim, ao receber um evento do tipo B_1 , a segunda condição é avaliada como verdadeira, tornando verdadeira, erroneamente, toda a regra de ativação referente ao tipo de situação *SituationC*. De forma análoga, supondo que a situação do tipo *SituationC* seja ativada pelos eventos B_1 e A_2 , a mesma não será considerada desativada ao receber um evento do tipo A_{2D} .

A plataforma SIMPLE monitora continuamente os fluxos de eventos a fim de verificar se as condições que levaram à ativação (ou desativação) de uma situação continuam satisfeitas, mantendo a integridade das mesmas. Além disso, cabe à plataforma SIMPLE impedir a execução de regras de ativação referentes às situações já ativas e a execução de regras de desativação referentes às situações já desativadas.

4.4 VISÃO GERAL E INSTRUÇÕES DE USO DA PLATAFORMA

Esta seção apresenta instruções de uso da plataforma SIMPLE, abstraindo das questões internas de implementação da plataforma. Ao longo desta seção, assume-se que o usuário possui a plataforma Esper corretamente configurada e funcional em seu ambiente. A Figura 37 ilustra uma visão geral da plataforma. O cliente (ou aplicação cliente) utiliza a plataforma SIMPLE por meio de três pacotes distintos: *Context Class Definitions*, *Situation Class Definitions* e *Situation Utility*. Cada um deles representa uma etapa distinta, no qual o usuário deve seguir de forma sequencial para utilizar a plataforma SIMPLE.

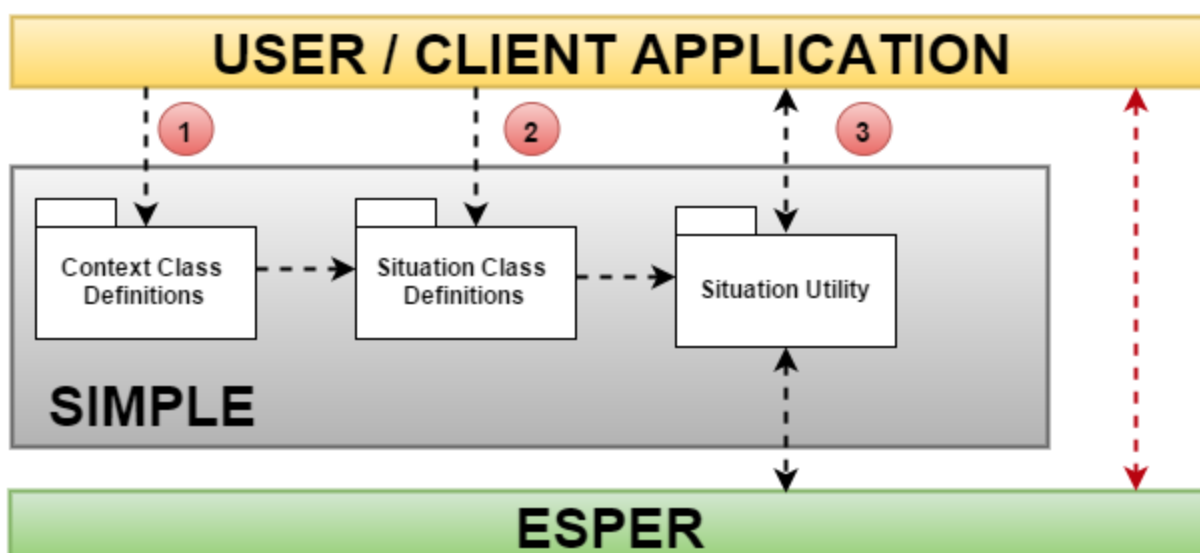


Figura 37. Visão geral da plataforma SIMPLE

4.4.1 Primeira Etapa

A primeira etapa consiste na especificação das classes Java, segundo o padrão POJO (*Plain Old Java Objects*), que implementam: (i) as entidades e relações contidas nos modelos de contexto (representados pelo pacote *Context Class Definitions*) e; os tipos de situação contidos nos modelos de situação (representados pelo pacote *Situation Class Definitions*).

Inicialmente, devem ser definidas as classes que implementam as entidades contidas no modelo de contexto. Assumindo o modelo de contexto ilustrado pela

Figura 8, a implementação da classe que representa a entidade Paciente é ilustrada pela Figura 38. Por simplicidade, os métodos *get* e *set* de cada atributo foram omitidos, porém necessários.

```
1 public class Patient extends Person{  
2  
3     private int temp;  
4     private String symptom;  
5     private String key;  
6     private boolean needHospitalization;  
7  
8 }
```

Figura 38. Implementação da classe Paciente no modelo de contexto da Figura 8

Além dos atributos contidos no modelo de contexto (*temp*, *symptom* e *needHospitalization*) é possível observar a ocorrência de atributo *key* (linha 5, Figura 38). Toda classe que representa uma entidade do modelo de contexto deve possuir o atributo *key*. Esse atributo, responsável por identificar unicamente cada elemento do domínio, deve ser atribuído manualmente pelo usuário. Considerando a classe Paciente, um possível valor para o atributo *key* é dado pelo seu número de CPF.

Uma vez criadas as classes que implementam as entidades contidas nos modelos de contexto, deve-se especificar as classes Java que implementam os tipos de situações definidos nos modelos de situação. A definição de tais classes devem seguir as recomendações apresentadas e discutidas na seção 4.2.

4.4.2 Segunda Etapa

Terminada a etapa de especificação dos tipos de situações, a plataforma SIMPLE possui todas as informações necessárias para gerenciar o ciclo de vida das situações especificadas. A segunda etapa consiste na inicialização da plataforma SIMPLE. A Figura 39 ilustra um diagrama de sequências com os passos necessários para inicialização e utilização correta da plataforma SIMPLE.

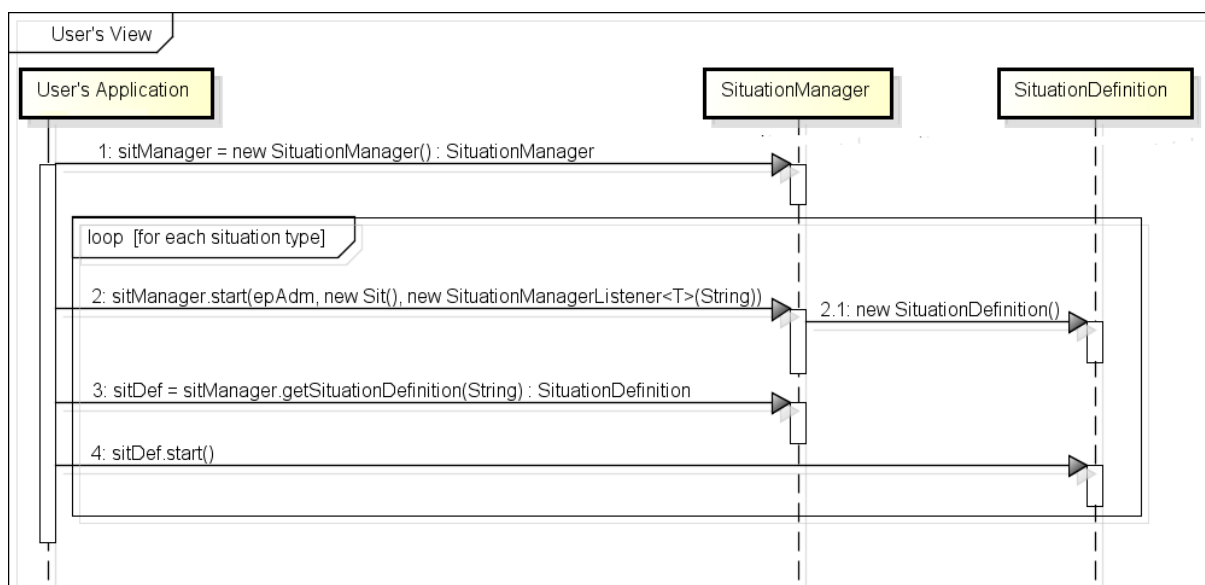


Figura 39. Diagrama de sequências dos passos necessários para utilizar a plataforma SIMPLE

A Figura 40, por sua vez, apresenta o trecho de uma aplicação que faz uso da plataforma SIMPLE para monitorar o tipo de situação *Fever*. As linhas 5, 6 e 7 da Figura 40 apenas iniciam as configurações iniciais da plataforma Esper, conforme explicado na seção 3.5. A inicialização da plataforma SIMPLE fica por conta das linhas 9, 10, 11 e 12 da Figura 40.

```

1 public class MainClass {
2     public static void main(String[] args){
3
4         /* Esper Configuration */
5         Configuration config = new Configuration();
6         EPServiceProvider epService = EPServiceProviderManager.getDefaultProvider(config);
7         EPAdministrator epAdm = epService.getEPAdministrator();
8
9         SituationManager sitManager = new SituationManager();
10        sitManager.start(epAdm, new Fever(), new SituationManagerListener<Fever>(Fever.class));
11        SituationDefinition fever = sitManager.getSituationDefinition("Fever");
12        fever.start();
13
14    }
15 }
  
```

Figura 40. Aplicação utilizando a plataforma SIMPLE para monitorar o tipo de situação *Fever*

O primeiro passo na utilização da plataforma SIMPLE consiste na criação de um objeto do tipo *SituationManager* (passo 1 na Figura 39 e linha 9 na Figura 40). Apenas uma instância do tipo *SituationManager* deve ser gerada e inicializada por domínio. Antes de iniciar a gerência dos ciclos de vida, cada tipo de situação deve

ser registrado no *SituationManager*, por meio do método *start* (passo 2 na Figura 39 e linha 10 na Figura 40).

Esse método possui três parâmetros. O primeiro deles consiste no *EPAdministrator* retornado pela plataforma Esper (linha 7, Figura 40). O segundo consiste num novo objeto do tipo de situação que deseja-se monitorar. O terceiro consiste em um *Listener* genérico disponibilizado pela plataforma SIMPLE. Por se tratar de uma estrutura genérica, durante a sua criação, deve ser especificado o tipo da situação a ser monitorada. Além disso, deve ser especificado o nome da classe Java que representa o tipo de situação como parâmetro do construtor.

Uma vez que o tipo de situação encontra-se devidamente registrado através do objeto do tipo *SituationManager*, é possível, então, iniciar a gerência do ciclo de vida daquele tipo de situação. Cada tipo de situação registrado possui um objeto do tipo *SituationDefinition* correspondente (passo 2.1, Figura 39), que é responsável por indicar o início do gerenciamento do ciclo de vida de cada tipo de situação. Para cada tipo de situação, o cliente deve solicitar ao *SituationManager* o respectivo *SituationDefinition*, através do método *getSituationDefinition* (passo 3 na Figura 39 e linha 11 na Figura 40). Para iniciar o gerenciamento do ciclo de vida do tipo de situação correspondente, basta invocar o método *start* desse objeto (linha 12, Figura 39).

O método *start* possui dois parâmetros opcionais (dois vetores de *Strings*). Esses parâmetros são utilizados para os casos nos quais as regras de ativação e desativação necessitem da especificação de parâmetros por parte do cliente. Por exemplo, o tipo de situação *Fever* pode ser restringido para um tipo de situação específico responsável por monitorar a febre de um único paciente. Nesse caso, para iniciar o ciclo de vida do tipo de situação, faz-se necessário especificar o nome do paciente a ser monitorado. A especificação do nome do paciente deve ser realizada por meio desses parâmetros. Nesse caso, ambos os parâmetros correspondem a um vetor unitário contendo uma única *String* com o nome do paciente.

Além disso, existe um terceiro parâmetro, também opcional, que é o *Listener* responsável por monitorar a ativação e desativação de um tipo de situação.

Conforme discutido anteriormente na seção 4.4.2, as ações executadas na ativação e desativação dos tipos de situação são implementadas, respectivamente, nos métodos *doActionAtAtivation* e *doActionAtDeactivation*. Porém, dependendo da complexidade exigida pela ação a ser executada, a simples sobrecarga desses métodos pode não ser suficiente. Nesses casos, o cliente deve implementar seu próprio *Listener* e especificá-lo como parâmetro no método *start*.

Para cada tipo de situação a ser gerenciado pela plataforma, devem-se repetir os passos 2, 3 e 4 da Figura 39, representados pelos códigos das linhas 10, 11 e 12 da Figura 40.

4.5 IMPLEMENTAÇÃO DA PLATAFORMA (VISÃO INTERNA)

A Figura 41 ilustra a visão geral da plataforma SIMPLE, de forma analoga à Figura 37, porém, exibindo os componentes internos do pacote *Situation Utility*. Conforme discutido na seção anterior, a plataforma SIMPLE é dividida em três pacotes: *Context Class Definition*, *Situation Class Definition* e *Situation Utility*. O pacote *Context Class Definition* contém as classes Java responsáveis por representar as entidades definidas no modelo de contexto. De forma análoga, o pacote *Situation Class Definition* contém as classes Java responsáveis por representar as entidades definidas no modelo de situações. Ambos os pacotes variam de acordo com o domínio em questão. É de responsabilidade do cliente implementar tais classes. Por sua vez, o pacote *Situation Utility* contém os arquivos implementados pelo presente trabalho. Os arquivos desse pacote possuem código fixo, independente de domínio, responsáveis por permitir a gerência das situações representadas pelos arquivos contidos no pacote *Situation Class Definition*.

Ao longo das seções seguintes são discutidos os componentes internos da plataforma SIMPLE contidos no pacote *Situation Utility*, ilustrados pela Figura 41. A explicação mais detalhada de cada componente, bem como trechos de códigos referentes às suas implementações, são apresentadas no Anexo A.

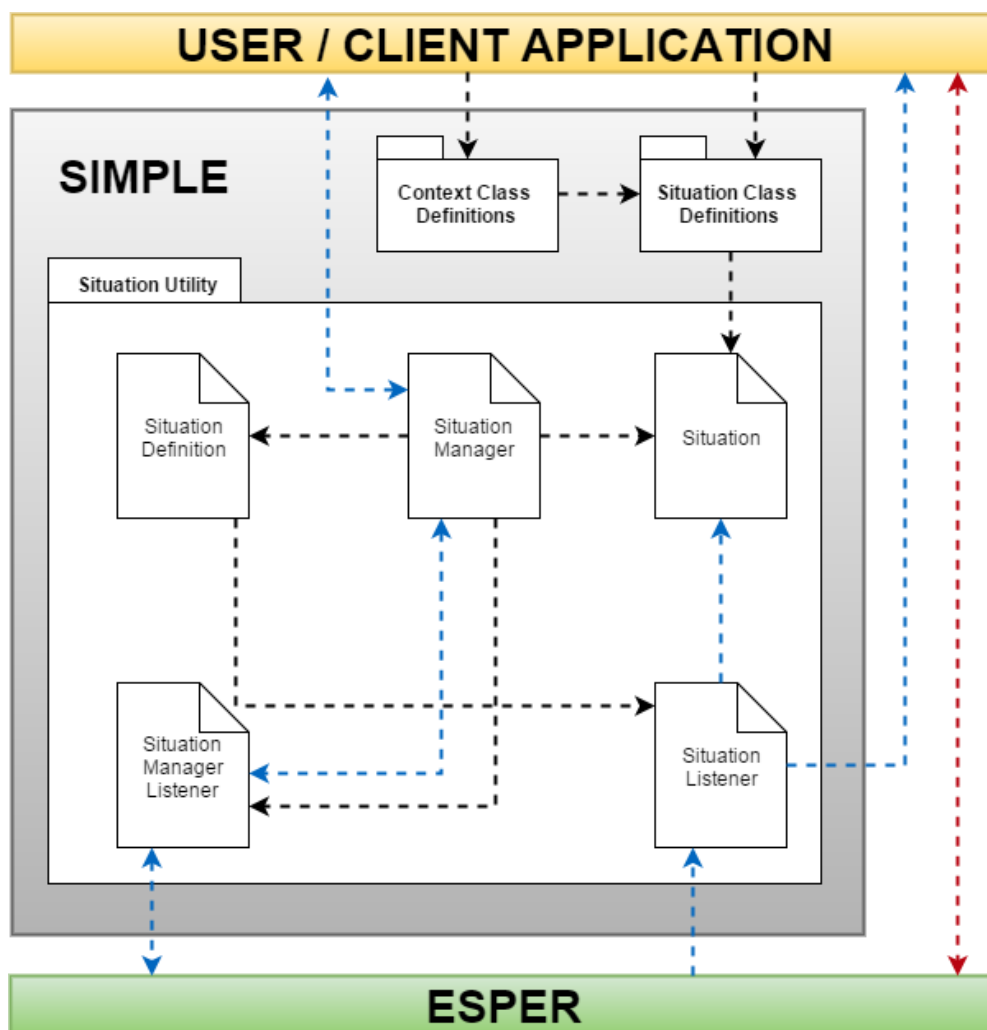


Figura 41. Visão Interna da Plataforma SIMPLE

As setas direcionais pontilhadas na Figura 41 possuem diferentes significados conforme suas respectivas cores. Setas na cor preta indicam dependência entre os componentes. Um exemplo é dado pela classe *SituationManager*, dependente da existência das estruturas de dados contidas nos componentes *Situation* e *SituationDefinition*. Setas na cor azul indicam interações entre os componentes, que podem ser dadas pela troca de eventos, como ocorre entre a plataforma Esper e os elementos *SituationManagerListener* e *SituationListener*. A seta na cor vermelha indica interação direta entre os componentes da camada cliente (representada por usuários ou aplicações clientes) e a plataforma Esper, sem a necessidade da plataforma SIMPLE como intermediária.

4.5.1 Situation

Conforme discutido na seção 4.2, o componente *Situation* consiste numa classe abstrata responsável por definir os atributos e métodos comuns aos tipos de situações. Toda classe que representa um tipo de situação (*i.e.*, classes do pacote *Situation Class Definitions*) devem estender esta classe. As informações contidas nos atributos desta estrutura são utilizadas pelos demais componentes. O componente *SituationManagerListener* (seção 4.5.4) utiliza a informações sobre o *status* (*i.e.*, ativada ou desativada) de cada instância de um tipo de situação, gerindo de forma adequada seus respectivos ciclos de vida. O componente *SituationDefinition* registra na plataforma Esper as regras de ativação e desativação contidas no tipo *Situation*, representadas por *queries* EPL.

4.5.2 SituationDefinition

O componente *SituationDefinition* permite que o usuário informe seu interesse em iniciar ou interromper a gerência do ciclo de vida de um dado tipo de situação, através de seu método *start*. Cada tipo de situação possui um objeto do tipo *SituationDefinition* correspondente, objetos estes armazenados no *SituationManager* (seção 4.5.3). Por meio dos objetos do tipo *SituationDefinition* as regras de ativação e desativação referente a cada tipo de situação são registradas na plataforma Esper e devidamente associadas ao *SituationManagerListener* (seção 4.5.4). O *SituationManagerListener*, juntamente com o *SituationManager*, são responsáveis por controlar o ciclo de vida das situações.

Em SIMPLE, sempre que uma regra de ativação ou desativação é acionada, é criado um evento, que corresponde a uma instância do tipo de situação em questão, indicando sua respectiva ativação ou desativação. Esses eventos correspondem, respectivamente, aos eventos dos tipos *Situation Creation Event* e *Situation Deactivation Event*, definidos no Modelo de Detecção de Situações em CEP (seção 4.1). Para cada tipo de situação, o objeto correspondente do tipo *SituationDefinition* registra uma regra responsável por detectar esses eventos. Por padrão, o *Listener* associado a essa regra é representado pelo *SituationListener* (seção 4.5.4). Se

necessário, o usuário pode implementar seu próprio *Listener* e substituí-lo pelo *SituationListener*, passando uma referência do *Listener* implementado ao invocar o método *start*.

4.5.3 SituationManager

O componente *SituationManager* é responsável por armazenar um objeto do tipo *SituationDefinition* (seção 4.5.2) para cada tipo de situação definido no pacote *Situation Class Definition*, de forma que o usuário possa manifestar seu interesse em iniciar ou interromper o gerenciamento do ciclo de vida dos mesmos. Além disso, esse componente armazena o último evento representando a ativação ou desativação de cada instância de um tipo de situação (*i.e.*, *Situation Creation Event* e *Situation Deactivation Event*). Conforme discutido na seção 4.1, através dessas instâncias armazenadas, a plataforma SIMPLE, por meio do componente *SituationManagerListener* (seção 4.5.4), é capaz de obter as informações necessárias para gerir corretamente o ciclo de vida de cada tipo de situação.

4.5.4 SituationListener e SituationManagerListener

O componente *SituationListener* consiste no *Listener* padrão utilizado pelo SIMPLE para monitorar os eventos de ativação e desativação dos tipos de situação especificados (*i.e.*, eventos do tipo *Situation Creation Event* e *Situation Deactivation Event*). Conforme discutido no capítulo 3, a plataforma Esper especifica que todo *Listener* deve implementar o método *update*, invocado quando um evento (ou um conjunto de eventos) satisfaz as restrições especificadas na *querie* EPL associada ao *Listener*. A implementação do método *update* para o *SituationListener* apenas exibe na tela uma mensagem de ativação, informando o tipo de situação associado ao mesmo, bem como seu identificador único (ID) e seus respectivos *start_time* e *end_time*. Se necessário, o usuário pode implementar seu próprio *Listener* e passar como parâmetro ao método *start* do objeto do tipo *SituationDefinition* correspondente.

Por sua vez, o componente *SituationManagerListener*, responsável pela gerência do ciclo de vida dos tipos de situações, consiste no *Listener* associado às *queries* EPL que representam as regras de ativação e desativação de cada tipo de situação (*i.e.*, eventos do tipo *Situation Trigger Creation Event* e *Situation Trigger Deactivation Event*). O seu método *update* é responsável por criar os respectivos eventos de ativação e desativação (*i.e.*, eventos do tipo *Situation Creation Event* e *Situation Deactivation Event*) referente a cada instância de um tipo de situação, e enviá-los à plataforma Esper.

Um evento correspondente a um *Situation Creation Event* deve ser criado caso a *query* associada ao *SituationManagerListener* corresponda a uma regra de ativação e a última instância do tipo de situação encontra-se desativada. A última instância de cada tipo de situação é obtida através de uma solicitação ao *SituationManager* (seção 4.5.3), no qual é possível verificar o *status* dessa instância. De forma análoga, um evento correspondente a um *Situation Deactivation Event* deve ser criado caso a *query* associada ao *SituationManagerListener* corresponda a uma regra de desativação e a última instância do tipo de situação encontra-se ativada. Os eventos criados pelo *SituationManagerListener* são armazenados no *SituationManager* como a nova última instância do respectivo tipo de situação, e enviados à plataforma Esper.

Caso nenhuma das duas condições anteriores seja satisfeita, é possível concluir que a *query* associada ao *SituationManagerListener*, necessariamente, indica (i) a ativação de um tipo de situação que já encontra-se ativado ou (ii) a desativação de um tipo de situação que já encontra-se desativado. Em ambos os casos, tal informação faz-se irrelevante ao usuário, levando ao término do método *update* sem a criação de um novo evento.

4.6 CRIAÇÃO DE UM EVENTO DE ATIVAÇÃO DE SITUAÇÃO UTILIZANDO A PLATAFORMA SIMPLE

Esta seção tem como objetivo ilustrar, por meio de um diagrama de sequências, a interação entre os componentes SIMPLE contidos no pacote *Situation*

Utility, durante o processo de criação de um evento representando a ativação de uma situação, ou seja, um evento do tipo *Situation Creation Event*. A Figura 42 apresenta o diagrama de sequência que ilustra a sequência de ações executadas a partir do momento que a função *update* do *SituationManagerListener* foi invocada.

Inicialmente, o *SituationManagerListener* recebe da máquina Esper, por meio da interface *EPRuntime*, um evento (ou conjunto de evento) que tornaram verdadeiras as restrições impostas por alguma *query* associada ao *SituationManagerListener* (passo 1, Figura 42). Como todas as *queries* correspondentes às regras de ativação e desativação são associadas ao *SituationManagerListener*, a priori, não se sabe se a *query* associada ao evento corresponde a uma regra de ativação ou desativação, e também não se sabe qual tipo de situação está associado à mesma. Uma vez que tais informações são inferidas, através de informações contidas no evento recebido pelo *SituationManagerListener*, é solicitado ao componente *SituationManager* a última instância gerada para o tipo de situação, através do método *getSituationAt* (passo 1.1, Figura 42).

A partir da última instância do tipo de situação, é obtido o seu status (ativada ou desativada). Caso a *query* esteja associada a uma regra de ativação e a última instância encontra-se desativada, segue-se para o próximo passo. De forma análoga, durante a criação de um evento do tipo *Situation Deactivation Event*, caso a *query* esteja associada a uma regra de desativação e a última instância encontra-se ativada, segue-se também ao próximo passo. Em caso negativo nas duas verificações, conforme discutido na seção 4.5.4, a informação recebida é irrelevante ao usuário final, e dessa forma a função *update* é interrompida.

No passo 1.2 da Figura 42, um evento correspondente à ativação de uma instância, ou seja, um *Situation Creation Event*, é criado, por meio do método *createNewSit* do componente *Situation*. Após criado o evento correspondente e seus atributos devidamente inicializados pelo *SituationManager* (*id*, *key*, *status*, *start_time*, *end_time*), é invocado o método *doActionAtCreateActivationEvent* do componente *Situation* (passo 1.3, Figura 42). Esse método, a princípio, não executa qualquer tipo de ação. Cabe ao usuário final sobrescrever tal método, caso seja de seu interesse executar algum tipo de ação nesse momento.

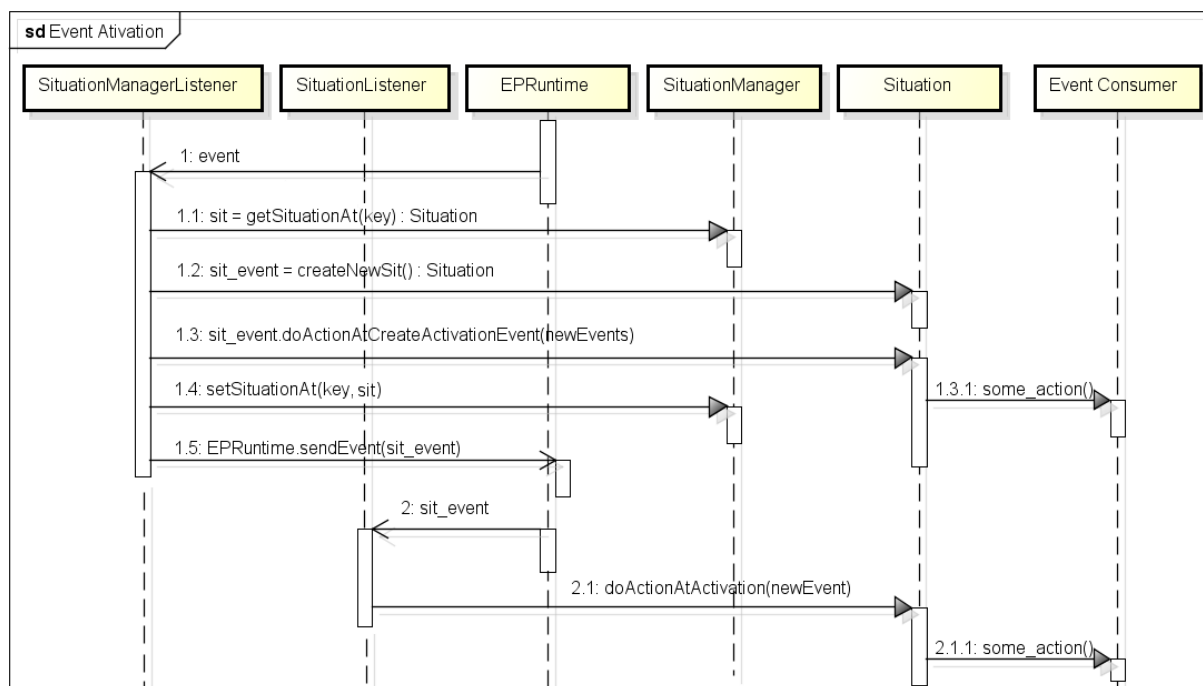


Figura 42. Diagrama de seqüências referente a criação de um *Situation Creation Event*

O novo evento criado é enviado ao componente *SituationManager* como a nova última instância do tipo de situação associado, através do método *setSituationAt* (passo 1.4, Figura 42). O evento criado é então enviado à máquina Esper por meio da função *sendEvent*, disponibilizada pela interface *EPRuntime* (passo 1.5, Figura 42). Como todo tipo de situação possui uma *query* correspondente associada ao *SituationListener*, o método *update* do mesmo é imediatamente invocado (passo 2, Figura 42).

Por fim, o usuário é notificado sobre a ocorrência de uma alteração no *status* do tipo de situação correspondente (*i.e.*, o tipo de situação foi ativado ou desativado), por meio do método *doActionAtActivation* do componente *Situation* (passo 2.1, Figura 42). Esse método executa a ação correspondente no cliente (*Event Consumer*). Por padrão, apenas uma mensagem é exibida, informando sobre tal ação (passo 2.1.1, Figura 42). Caso seja de interesse, o usuário pode implementar uma ação mais complexa (o método pode ser sobrecarregado), ou um *Listener* implementado pelo próprio usuário pode ser associado à *query* em questão durante a chamada do método *start* do componente *SituationDefinition*.

O processo de criação de um evento do tipo *Situation Deactivation Event* é análogo ao processo aqui apresentado. As diferenças ficam por conta dos passos

1.2 e 2.1, nos quais os métodos (i) *createNewSit* e (iii) *doActionAtActivation* são substituídos, respectivamente, pelos métodos (i) *doActionAtCreateDeactivationEvent* e (ii) *doActionAtDeactivation*. Além disso, o passo 1.3 é removido do fluxo de ações.

4.7 CONSIDERAÇÕES DO CAPÍTULO

Este capítulo apresentou a plataforma SIMPLE, responsável por detectar e gerenciar o ciclo de vida das situações usando a plataforma Esper como base. Foram discutindo detalhes gerais sobre a plataforma, dividida em duas visões. Primeiramente foi apresentada a visão do usuário, discutindo detalhes gerais sobre a utilização da plataforma SIMPLE para detectar e gerenciar o ciclo de vida das situações de interesse. Posteriormente, foi apresentada uma visão interna, discutindo detalhes a respeito do funcionamento dos componentes internos da plataforma SIMPLE.

Além disso, foram apresentadas as estruturas de dados necessárias para a definição de um tipo de situação utilizando a plataforma SIMPLE, de forma a permitir o gerenciamento adequado de seu ciclo de vida.

O capítulo seguinte apresenta os padrões, identificados durante a elaboração deste trabalho, responsáveis pela especificação das regras de ativação e desativação de cada tipo de situação. Além disso, são apresentadas as regras de transformações desenvolvidas com base nos padrões propostos.

5 REGRAS DE TRANSFORMAÇÃO

Este capítulo tem como objetivo apresentar e discutir as principais regras de transformação utilizadas na geração de regras EPL a partir de modelos SML (seção 2.4). Ao longo das seções seguintes são apresentados quatro padrões, responsáveis por cobrir as principais regras de transformação. Um padrão representa um tipo de situação modelado e origina duas regras EPL: (i) uma responsável pela ativação das instâncias do tipo de situação e (ii) uma responsável pela desativação das instâncias do tipo de situação. Os padrões desenvolvidos são ilustrados por meio de duas figuras: uma com a regra de transformação de ativação e a outra com a regra de transformação de desativação do padrão modelado em SML.

Cada figura apresenta um modelo SML e sua respectiva regra EPL (de ativação ou desativação). Relações entre os elementos de SML e sua respectiva correspondência textual (em EPL) são representadas por linhas pontilhadas direcionais. A fim de facilitar o entendimento e visualização dos padrões apresentados, algumas palavras encontram-se coloridas nas cores vermelha, azul ou verde. Palavras coloridas na cor vermelha indicam palavras reservadas pela linguagem Esper EPL (ex: *select*, *from*). Palavras coloridas na cor azul, indicam campos gerados pela implementação do SIMPLE (ex: *key*, *activated*). Por fim, palavras coloridas na cor verde, indicam janelas temporais (ex: *win:timer*).

Este capítulo está organizado da seguinte forma: as Seções 5.1, 5.2, 5.3 e 5.4 apresentam os quatro principais padrões utilizados pelo presente trabalho: (i) padrão Entidade-Contexto Intrínseco; (ii) Padrão Entidade-Contexto Relacional; (iii) *Exists* e (iv) Padrão Situação composta; respectivamente; e, por fim, a Seção 5.5 apresenta as considerações do capítulo.

5.1 PADRÃO ENTIDADE-CONTEXTO INTRÍNSECO

O primeiro padrão, denominado “Entidade-Contexto Intrínseco”, é ilustrado pelas figuras Figura 43 e Figura 44, que apresentam, respectivamente, as regras de transformação de ativação e desativação. Esse padrão apresenta o mapeamento referente a um tipo de situação que contém uma entidade (*Entity*), um de seus atributos (*attribute*) e uma relação formal (*relation*) entre esse atributo e um valor (*value*).

Na Figura 43 também é possível observar a ocorrência de um losango azul, especificando um *binding name*. O *binding name* pode ser entendido como um apelido atribuído a uma entidade, como ocorre com a entidade *Entity* da Figura 43. O *binding name* é útil na definição de uma situação composta por mais de uma entidade de um mesmo tipo. Através de seus respectivos *binding names* é possível diferenciar cada entidade.

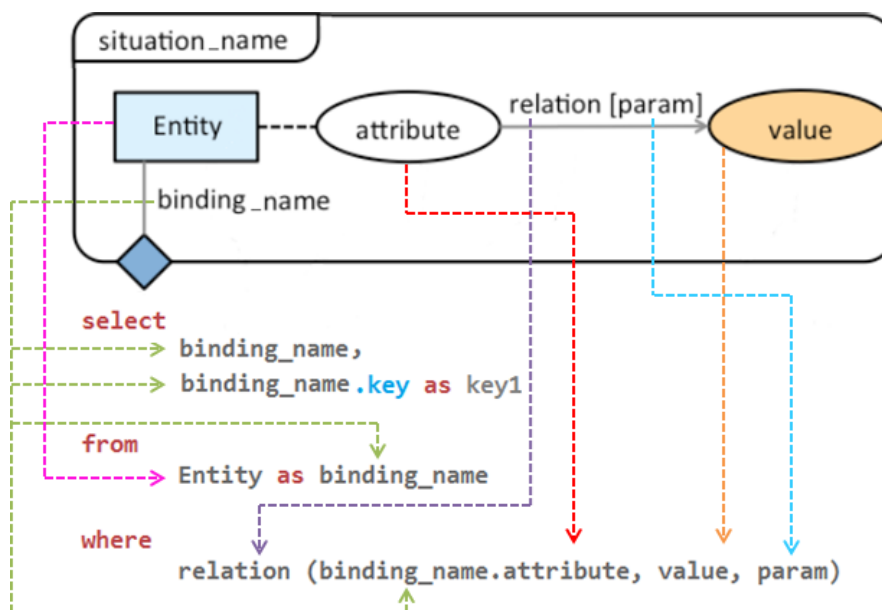


Figura 43. Regra de transformação de ativação do padrão Entidade Contexto-Intrínseco

Entidades ditas participantes da situação, assim como ocorre com a entidade *Entity* da Figura 43, são mapeadas como uma *view* na cláusula *from*. Como foi explicitado um *binding name*, o mesmo é atribuído na *view* por meio da palavra “as”. Toda entidade utilizada na cláusula *from* deve ter seu nome (ou *binding name*

associado, como ocorre na Figura 43) apresentado na cláusula *select*, juntamente com seu respectivo atributo *key*.

Conforme discutido na seção 4.4, o atributo *key* é utilizado pelo SIMPLE para identificar unicamente cada elemento no domínio, devendo ser atribuído pelo usuário. Por exemplo, caso a entidade represente uma pessoa, um possível valor para o atributo *key* é dado pelo número de seu cpf. Através do atributo *key* é possível identificar diferentes instâncias do mesmo tipo de situação. Valores distintos de *key* indicam instâncias de situações distintas e devem, portanto, ter seus ciclos de vida gerenciados de forma independente. Atributos *key* recebem um *binding name* definido pela palavra *key* concatenada a um contador sequencial.

O conjunto formado por atributo, valor e relação formal é mapeado numa restrição na cláusula *where*. Essa restrição é representada por uma função, na qual o atributo e o valor são passados como parâmetros, bem como possíveis parâmetros (*param*) especificados na relação. Nesse caso, assume-se que o cliente possui uma função implementada de mesmo nome em sua aplicação, sendo aceitos o mesmo número e tipos de parâmetros mapeados. Além disso, essa função deve retornar um valor do tipo *boolean* (*i.e.*, *true* ou *false*). Caso a relação formal seja uma relação primitiva, por exemplo, a relação de igualdade (*equals*), o mapeamento é realizado de forma direta, ou seja, através da restrição “atributo = valor”.

O nome da situação (*situation_name*) não é utilizado no processo de mapeamento apresentado pela Figura 43. O mesmo ocorre nos demais mapeamentos referentes às regras de transformação de ativação apresentadas ao longo das seções seguintes. O nome da situação é utilizado apenas no mapeamento referente às regras de transformação de desativação.

A Figura 44 ilustra a regra de transformação de desativação correspondente ao mesmo padrão apresentado pela Figura 43. De forma análoga ao que ocorre com os participantes na regra de transformação de ativação, na regra de transformação de desativação o nome da situação é utilizado na cláusula *select* e na cláusula *from*. Porém, na cláusula *from*, não é necessário utilizar o atributo *key*, pois os participantes da situação identificam unicamente uma instância de um tipo da situação, e não a situação em si.

Na cláusula *from* (trecho 2 da Figura 44) o nome da situação é associado a uma janela: “*std:unique(id)*”, indicando a criação uma janela diferente para cada situação com atributo *id* diferente. O atributo *id* é criado e atribuído a todas as instâncias de algum tipo de situação, pelo SIMPLE, no momento de sua ativação. Seu papel é semelhante ao do campo *key*, identificando unicamente cada instância de algum tipo de situação. Esse campo é necessário para garantir a integridade das situações, uma vez que, para um mesmo tipo de situação, distintas instâncias podem existir ao longo do tempo. Cada instância de um tipo de situação é definida pela existência de um evento de ativação e, eventualmente, um evento de desativação. A utilização do atributo *id* permite associar os eventos de ativação e desativação correspondentes, para todas as instâncias geradas. A *view* criada pelo nome da situação e sua respectiva janela recebem um *binding name* idêntico ao nome da situação.

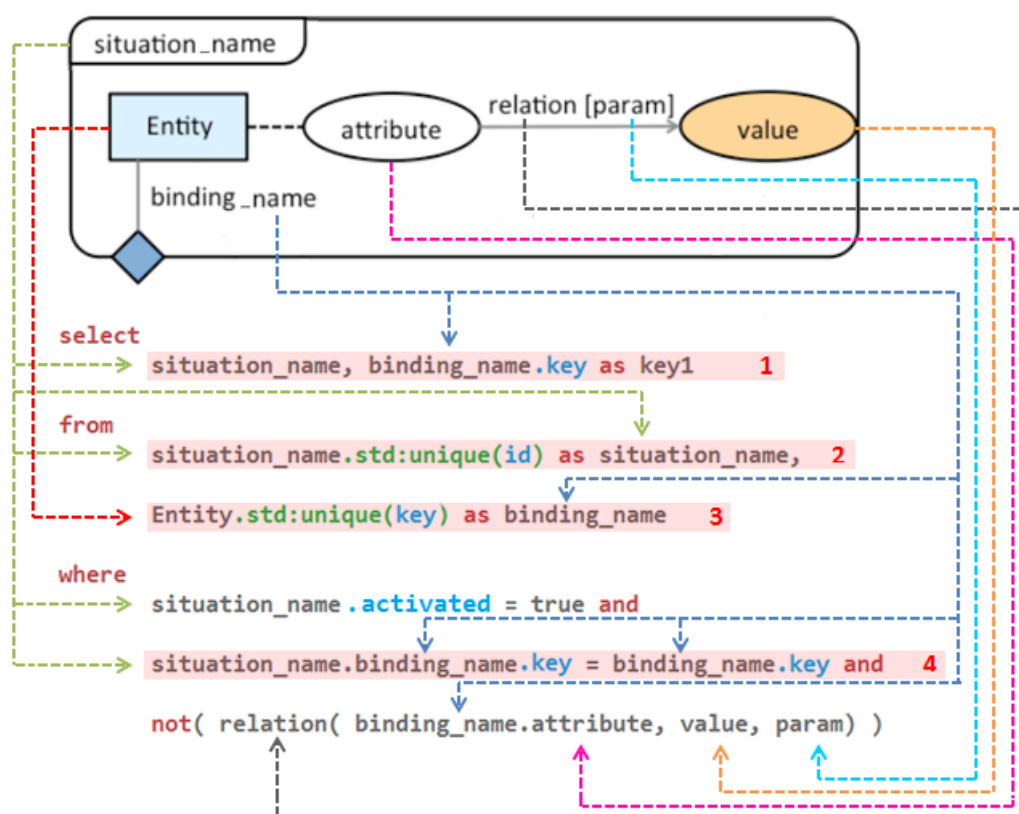


Figura 44. Regra de transformação de desativação do padrão Entidade-Contexto Intrínseco

A transformação gera uma restrição na cláusula *where* usando o nome da situação, para verificar se a situação encontra-se ativa. Assim como na regra de transformação de ativação (Figura 43), a entidade *Entity* é utilizada nas cláusulas

select e *from*. Na cláusula *select* (trecho 1 da Figura 44), a entidade é associada ao respectivo atributo *key*, enquanto na cláusula *from* (trecho 3 da Figura 44), a entidade é associada ao respectivo *binding name*. A principal diferença é dada pelo uso de uma janela associada: “*std:unique(key)*”, indicando a necessidade de criar uma janela diferente para cada entidade com *key* diferente. Caso o participante seja uma situação participante, deve-se trocar o atributo *key* pelo atributo *id*.

Uma diferença importante entre as regras de transformação de ativação e desativação pode ser observada no trecho 4 da Figura 44. Toda situação mantém referências para os participantes que levaram à sua ativação. Dessa forma, os participantes capturados pela cláusula *from* da regra de desativação têm seu atributo *key* comparado ao respectivo atributo referente ao participante de mesmo tipo armazenado durante a ativação da situação. Isso permite verificar se a entidade capturada pela cláusula *from* representa o mesmo participante (ou instância da mesma situação participante) que levou à ativação da situação em questão. A restrição gerada na cláusula *where* segue os princípios da regra de transformação de ativação (Figura 43), porém, o padrão gerado deve ser negado, através do operador *not*.

5.2 PADRÃO ENTIDADE-CONTEXTO RELACIONAL

O segundo padrão, denominado “Entidade-Contexto Relacional”, é representado pelas figuras Figura 45 e Figura 46, que ilustram, respectivamente, as regras de transformação de ativação e desativação. Esse padrão apresenta o mapeamento referente a um tipo de situação que contém duas entidades (*EntityA* e *EntityB*) relacionados por um contexto relacional (*Relation*), por meio das relações *relationA* e *relationB*. Como discutido na seção 2.4, um contexto relacional representa uma relação entre duas entidades. O contexto relacional existe apenas enquanto a relação entre as entidades relacionadas existem. Na cláusula *from*, não é necessário especificar as entidades relacionadas pelo contexto relacional, bastando especificar uma única *view* para o contexto relacional (o contexto relacional já possui referências para as suas entidades relacionadas).

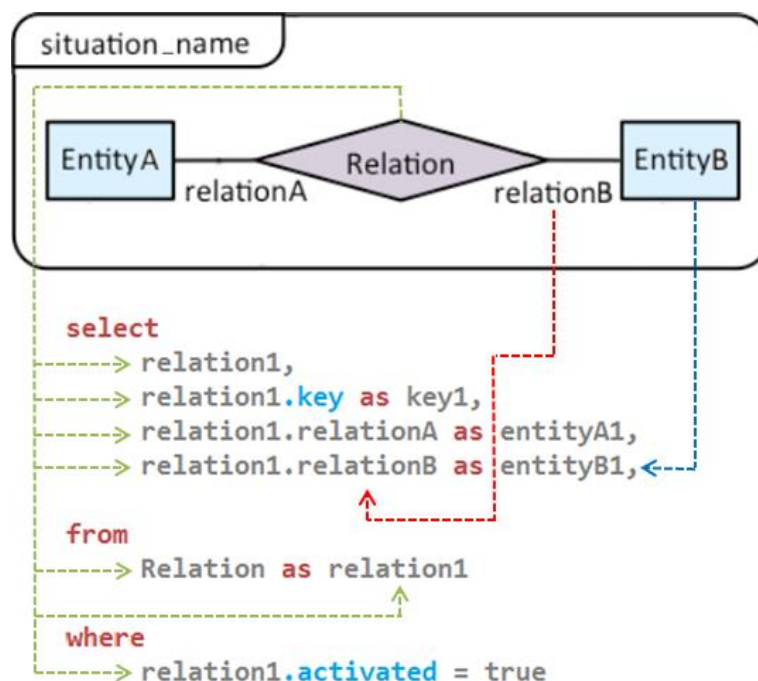


Figura 45. Regra de transformação de ativação do padrão Entidade-Contexto Relacional

Assim como na regra de transformação de ativação do padrão “Entidade-Contexto Intrínseco” (Figura 43), por ser um participante, o contexto relacional deve ter seu nome e seu atributo *key* retornado pela cláusula *select*. Da mesma forma, o contexto relacional deve também constituir uma *view* na cláusula *from*. Como nesse padrão não foi explicitado um *binding name*, ele é gerado automaticamente pela transformação. O *binding name* é formado pela concatenação do nome da *view* (*Relation*), iniciado com letra minúscula, e um contador sequencial, único para cada tipo diferente de entidade, iniciando sempre com o valor 1. Neste exemplo, o *binding name* resultante é *relation1*, conforme ilustrado pela Figura 45.

Para as entidades *EntityA* e *EntityB*, apesar de representarem entidades participantes, não é necessário retornar seus atributos *key* na cláusula *select*, pois o contexto relacional representa uma relação única entre tais entidades, refletindo tal comportamento na semântica de seu atributo *key*. Nesse caso, devem-se retornar apenas seus respectivos nomes na cláusula *select*. Assim como para a entidade do tipo *Relation*, essas entidades recebem um *binding name* gerado automaticamente pela concatenação do nome da *view*, iniciado com letra minúscula, e um contador sequencial, tendo como resultado os nomes *entityA1* e *entityB2*. Por simplicidade, as

setas direcionais ilustram o mapeamento apenas da entidade do tipo *EntityB*, uma vez que tal mapeamento é análogo para a entidade do tipo *EntityA*.

O contexto relacional impõe uma restrição sobre seu atributo *activated* na cláusula *where*. O atributo *activated*, para um contexto relacional, indica se num dado momento do tempo, a relação existe ou não; seu valor igual à *true* indica a existência da relação entre duas entidades. Quando essa relação deixa de existir, o atributo *activated* do contexto relacional tem seu valor igual a *false*. Assim, para regra de ativação, o seu atributo é sempre comparado ao valor *true*; enquanto que, para regra de desativação, é sempre comparado ao valor *false*.

Figura 46 ilustra a regra de transformação de desativação para este padrão segundo os critérios discutidos pela regra de transformação de desativação do padrão “Entidade-Contexto Intrínseco” (Figura 44). O nome da situação é utilizado na cláusula *select*, sem o atributo *key*, e na cláusula *from*, acrescida da janela “*std:unique(id)*”. Além disso, o nome da situação impõe uma restrição na cláusula *where*, verificando se a mesma encontra-se ativa por meio do atributo *activated*.

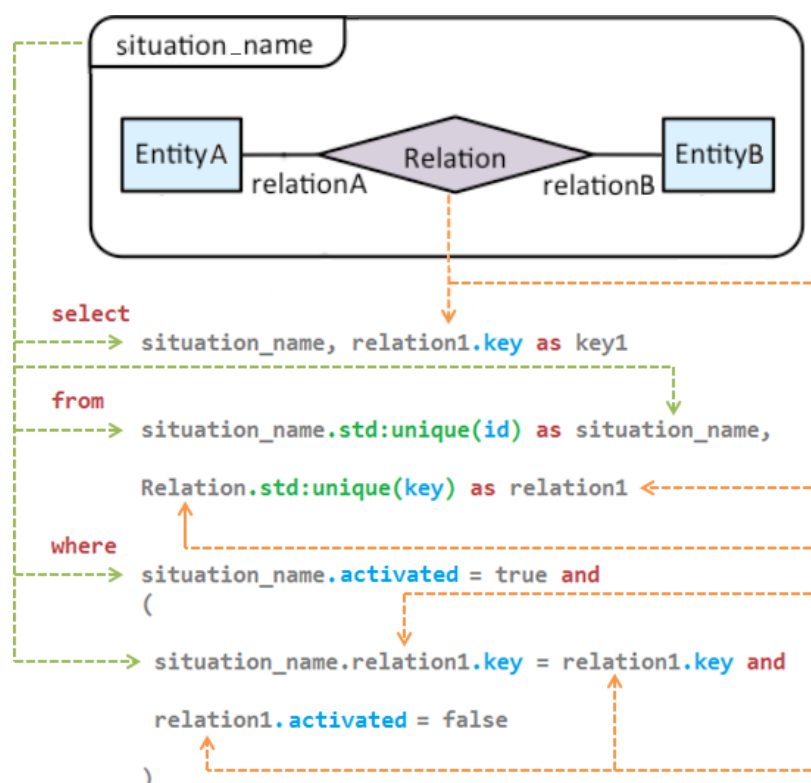


Figura 46. Regra de transformação de desativação do padrão Entidade-Contexto Relacional

Da mesma forma, por ser uma entidade participante, a entidade *Relation* é utilizada na cláusula *select*, com o atributo *key*, e na cláusula *from*, acrescida da janela “*std:unique(key)*”. Novamente, como não foi explicitado um *binding name*, o mesmo é gerado automaticamente pela transformação, resultando no *binding name relation1*.

A diferença é dada pelo atributo *activated*, na restrição da cláusula *where*, sendo seu valor comparado a *false* (encontra-se negado) e não mais ao valor *true*, como na regra de transformação de ativação. Assim como ocorre na regra de desativação do padrão “Entidade-Contexto Intrínseco” (Figura 44), é necessário comparar o atributo *key* do participante *Relation* ao participante de mesmo tipo armazenado durante a ativação da situação.

5.3 PADRÃO EXISTS

O terceiro padrão, denominado “*Exists*”, é representado pelas figuras Figura 47 e Figura 48, que ilustram, respectivamente, as regras de transformação de ativação e desativação. Este padrão apresenta um tipo de situação composta que contém uma situação participante (*SituationA*), um de seus atributos (*final_time*) e uma relação formal (*within the past*) entre esse atributo e um valor (*value*). O retângulo de borda arredondada representando a situação participante está na cor branca, indicando que a ocorrência de uma situação passada. Além disso, é aplicado o operador existencial \exists (*exists*) sobre a situação participante *SituationA* (chamada aqui de situação existencial).

O operador existencial indica o interesse em qualquer situação do tipo *SituationA*, em um determinado período de tempo. Dessa forma é necessário especificar uma janela temporal, com o intuito de indicar por quanto tempo deve-se aguardar pela não ocorrência de uma situação do tipo *SituationA* e, assim, desativá-la. Em SML, a especificação de uma janela temporal é realizada através dos atributos *initial_time* e *final_time* relacionados a um valor por meio da relação formal *within the past*.

A particularidade encontrada no uso do operador existencial é refletida em sua regra de transformação. Assim como nos demais padrões, a situação participante *SituationA* é utilizada na cláusula *from* associada a um *binding name*. Como o operador existencial abrange diversas instâncias diferentes do mesmo tipo de situação, apenas o *binding name* da situação é utilizado na cláusula *select*, sem o respectivo atributo *key*.

Para uma situação existencial, a janela temporal é apenas utilizada na regra de desativação (diferente do padrão discutido na seção 5.2). Na cláusula *where*, apenas é verificado se a primeira ocorrência da situação do tipo *SituationA* está desativada (ou ativada, caso a situação participante possua coloração cinza no modelo SML).

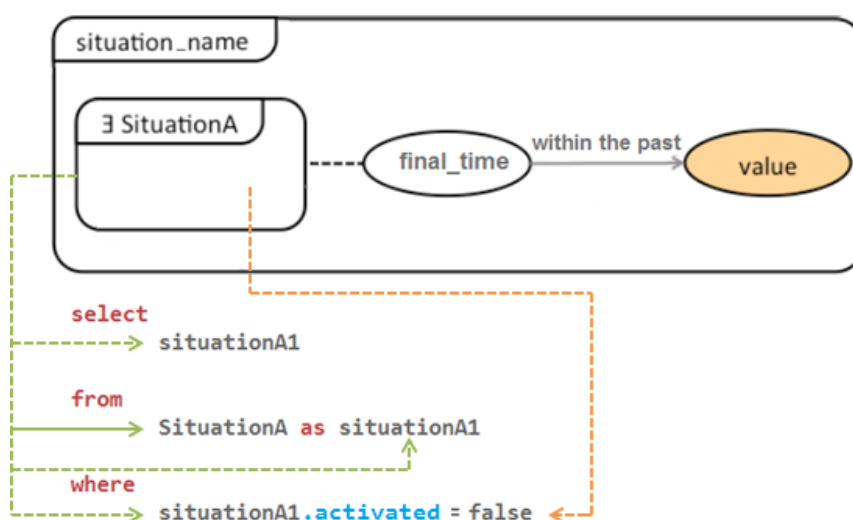


Figura 47. Regra de transformação de ativação do padrão *Exists*

Devido as particularidades do padrão “*Exists*”, diferenciando em muito dos demais tanto na regra de transformação de ativação quanto na regra de desativação, não foi possível generalizar este padrão de forma a permitir sua utilização na composição com os demais padrões apresentados. Assim, foi definido que ao utilizar uma situação existencial na composição de uma situação, a situação deve limitar-se a apenas um participante (nesse caso, a própria situação existencial). Caso seja necessário definir uma situação a partir da ocorrência de múltiplas situações existenciais, deve-se definir situações distintas, cada uma contendo uma das situações existenciais, e, então, utilizá-las como situações participantes de uma nova situação.

A Figura 48 ilustra a regra de transformação de desativação correspondente ao mesmo padrão apresentado pela Figura 47. O nome da situação é utilizado na cláusula *select*, assim como nos demais padrões apresentados. Na cláusula *from* é utilizado o operador *pattern* (seção 3.4). Além disso, durante a especificação de uma *view* utilizando o operador *pattern*, é possível relacionar as *views* por meio de operadores, como ocorre na Figura 6 por meio do operador *followed by* (seção 3.4.1), representado por pelo símbolo “->”. O operador *followed by* impõe uma ordem, indicando que a expressão do lado direito será analisada apenas após a expressão do lado esquerdo ser avaliada como *true*.

A regra transformação ilustrada pela Figura 48 especifica que a situação deve ser desativada somente quando não existir um evento de desativação referente a uma situação do tipo *SituationA*. A não ocorrência de um evento deve estar associada a um intervalo, conforme discutido na seção 3.4.1. Esse intervalo é representado por uma janela temporal, especificada por meio da restrição formada (i) pelo atributo *final_time*, (ii) a relação *within the past* e (ii) o literal *value*. Essa restrição é mapeada na cláusula *from* através o operador “*timer:interval(value)*”. O operador *timer* funciona como um contador, conforme discutido na seção 3.4.1. Como essa regra de transformação corresponde a uma regra de desativação, o operador *timer* deve iniciar sua contagem apenas após a situação ter sido ativada. Para representar tal comportamento é necessário utilizar o operador *followed by*, indicando para que a contagem deva iniciar somente após a ocorrência de um evento de ativação referente ao tipo de situação em questão. Isso faz-se necessário pois o Esper inicia as contagens dos tempos no momento em que a regra EPL é registrada. Sem o operador *followed by*, caso a situação demore um tempo maior que *value* para ser ativada, a regra é desativada imediatamente após sua ativação.

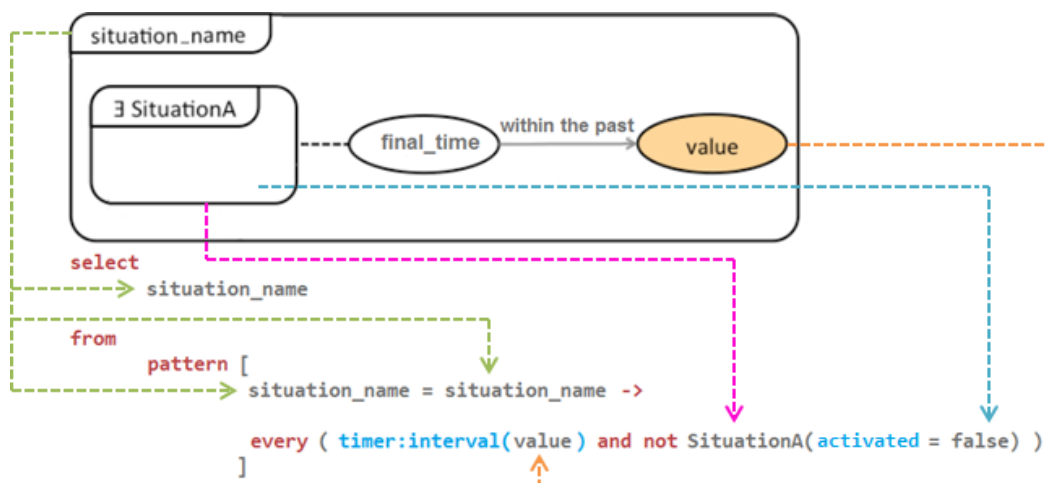


Figura 48. Regra de transformação de desativação do padrão *Exists*

Com o auxílio do operador *every* (seção 3.4.2), é possível detectar todos os intervalos temporais nos quais não ocorre um evento de desativação do tipo de situação especificado. Sem o operador *every*, o operador *pattern* verificaria apenas a primeira ocorrência do padrão especificado.

5.4 PADRÃO SITUAÇÃO COMPOSTA

O quarto e último padrão, denominado “Situação Composta” é representado pelas figuras Figura 49 e Figura 50, que ilustram, respectivamente, as regras de transformação de ativação e desativação. Esse padrão apresenta o mapeamento referente a um tipo de situação composta que contém mais de uma situação participante (*SituationA* e *SituationB*), relacionadas por atributos de suas respectivas entidades participantes (*EntityA* e *EntityB*). Com o intuito de prover uma melhor visualização, as Figura 49 e Figura 50 apresentam setas direcionais indicando apenas os mapeamentos do tipo de situação *SituationA*, uma vez que os mapeamentos para o tipo de situação *SituationB* são análogos.

As entidades *SituationA* e *SituationB*, uma vez que desempenham papel de participantes, devem ser utilizadas na cláusula *select*, juntamente com seus respectivos atributos *key*. Seus atributos, relacionados pela relação formal *relation* geram uma restrição na cláusula *where*. Assim como ocorre no padrão “Entidade-Contexto Relacional” (seção 5.2), para as entidades *EntityA* e *EntityB* não é

necessário retornar seus atributos *key* na *cláusula select*, apenas seus respectivos nomes. Isso ocorre por estas entidades comporem os tipos de situação *SituationA* e *SituationB*, respectivamente, nos quais já possuem seus atributos *key* retornados na *cláusula select*. A transformação deste padrão utiliza três operadores da linguagem Esper EPL para geração da *cláusula from*: *pattern*, *every* e *followed by*.

O operador *pattern* foi utilizado conforme explicado na regra de transformação de desativação do padrão “*Exists*” (Figura 48). A situação ilustrada pela Figura 49 deve ser ativada para cada par de ocorrências de situações dos tipos *SituationA* e *SituationB* (respeitando a restrição definida entre os atributos das entidades *EntityA* e *EntityB*). Inicialmente, deseja-se detectar a ativação referente a uma situação do tipo *SituationA* (*i.e.*, detectar um evento Esper que satisfaça a condição de ativação especificada no tipo de situação *SituationA*). O código correspondente pode ser observado no trecho 1 da Figura 49. Após detectada a ativação de uma situação do tipo *SituationA*, deve-se aguardar pela ativação referente a uma situação do tipo *SituationB*. Para que o padrão especificado represente o comportamento esperado, a situação do tipo *SituationA* deve continuar ativa (*i.e.*, não deve existir um evento de desativação) até a detecção da ativação de uma situação do tipo *SituationB*. Para representar esse comportamento, foi utilizado o operador *followed by* (“->”) combinado com o operador *not* para especificar a não ocorrência da desativação da situação entre os eventos de ativação, ilustrado no trecho 2 da Figura 49.

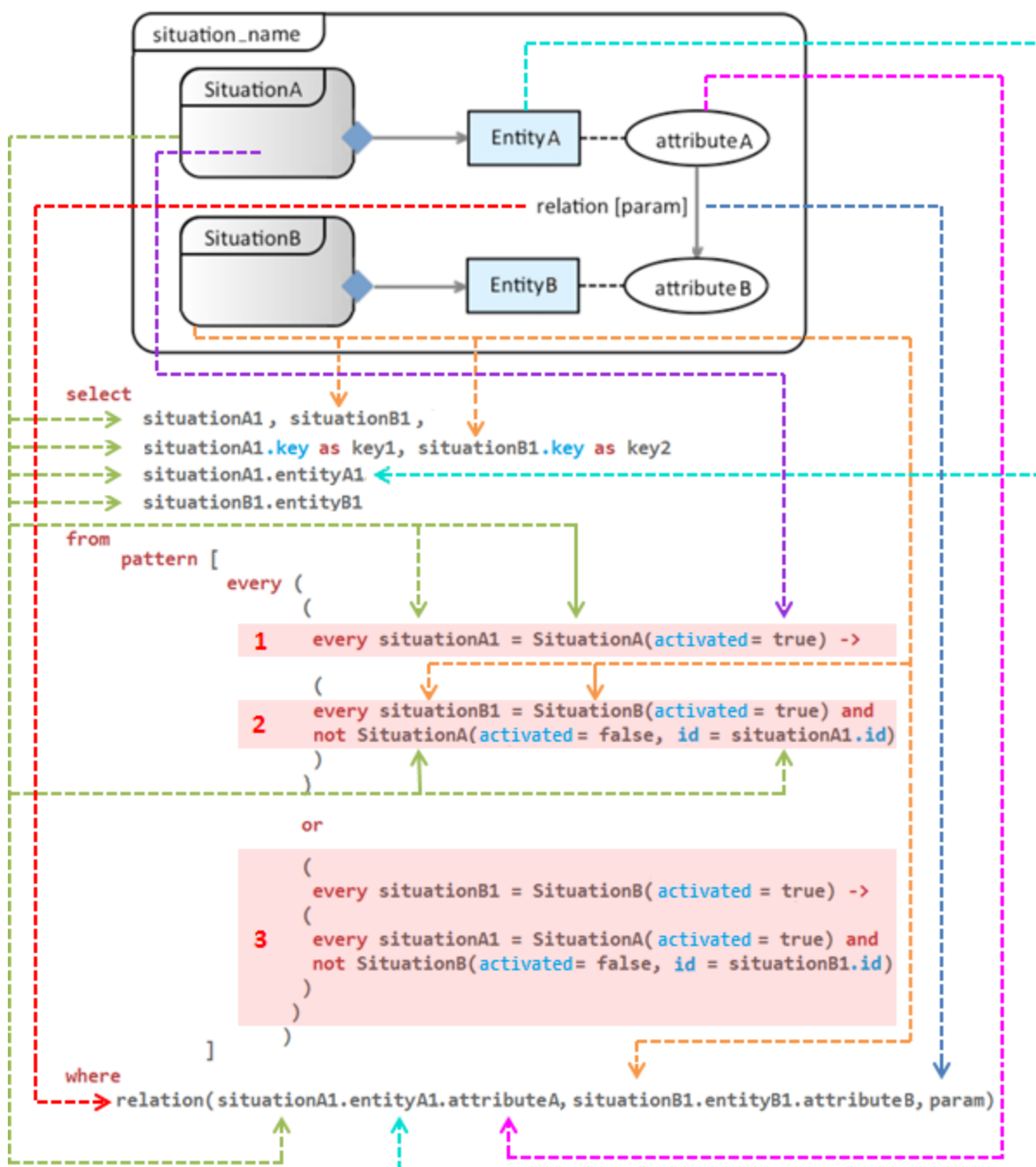


Figura 49. Regra de transformação de ativação do padrão Situação Composta

No modelo SML da Figura 49, a ordem de detecção das situações não é importante. Portanto, a especificação da situação em EPL também deve detectar e ativar a situação composta caso uma situação do tipo *SituationB* seja ativada antes de uma situação do tipo *SituationA*. Porém, conforme discutido na seção 5.3, o operador *followed by* impõe uma ordem. Dessa forma, é necessário especificar um trecho 3, análogo à combinação dos trechos 1 e 2, onde são realizadas as mesmas

verificações anteriores, porém assumindo dessa vez a possibilidade de ser detectado uma situação do tipo *SituationB* antes de uma situação do tipo *SituationA*.

Isso leva a necessidade de especificar $N!$ combinações, onde N representa o número de participantes das situações. Para dois participantes, apenas duas combinações são necessárias. Porém, esse número cresce rapidamente. Para cinco participantes, por exemplo, são necessárias 120 combinações. A especificação manual desse tipo de padrão torna-se inviável, justificando sua geração automática através do uso de técnicas de desenvolvimento orientado a modelos.

O operador *every* faz-se necessário, pois, por padrão, ao utilizar o operador *pattern*, apenas a primeira ocorrência do padrão é detectada. A utilização do operador *every* indica que devem ser detectadas todas as ocorrências possíveis desse padrão. Na Figura 49, isso é realizado pelo primeiro e mais externo operador *every* encontrado no padrão. Internamente, nos trechos 1 e 2 da Figura 49, também é possível observar o uso do operador *every*. Nesses casos, o operador indica que, mesmo para um mesmo tipo de situação, é desejado monitorar todas as instâncias desse mesmo tipo. Por exemplo, várias instâncias do tipo *SituationA* podem ter sido ativadas e desativadas até o momento em que uma situação que encontra-se ativa preceda uma situação do tipo *SituationB*, sem a existência de uma desativação da situação do tipo *SituationA* entre os mesmos. Sem o operador *every*, apenas a primeira ocorrência desse tipo seria analisada, levando à não detecção do padrão para todos os pares de instâncias (*SituationA-SituationB* e *SituationB-SituationA*).

A Figura 50 ilustra a regra de transformação de desativação correspondente ao mesmo padrão apresentado pela Figura 49. Essa regra de transformação segue os mesmos critérios discutidos na regra de transformação de desativação dos padrões “Entidade-Contexto Intrínseco” (Figura 44) e “Entidade-Contexto Relacional” (Figura 46).

O nome da situação é utilizado na cláusula *select*, sem o atributo *key*, e na cláusula *from*, acrescida da janelas “*std:unique(id)*”. Além disso, o nome da situação dá origem a uma restrição na cláusula *where*, verificando se a mesma encontra-se ativa, através do atributo *activated*. Uma vez que as situações *SituationA* e *SituationB* são situações participantes, as mesmas são utilizadas na cláusula *select*,

com o atributo *key*, e na cláusula *from*, acrescida da janela “*std:unique(id)*”. Nesse caso a janela utiliza o campo *id* no lugar do campo *key* por se tratarem de situações participantes. Como não foi explicitado um *binding name*, os mesmos são gerados automaticamente pela transformação, resultando nos *binding names* *situationA1* e *situationB1*. É importante observar a utilização do nome da situação precedendo os nomes das situações participantes na cláusula *select*, indicando que devem ser retornados os exatos eventos que levaram à ativação do tipo de situação em questão.

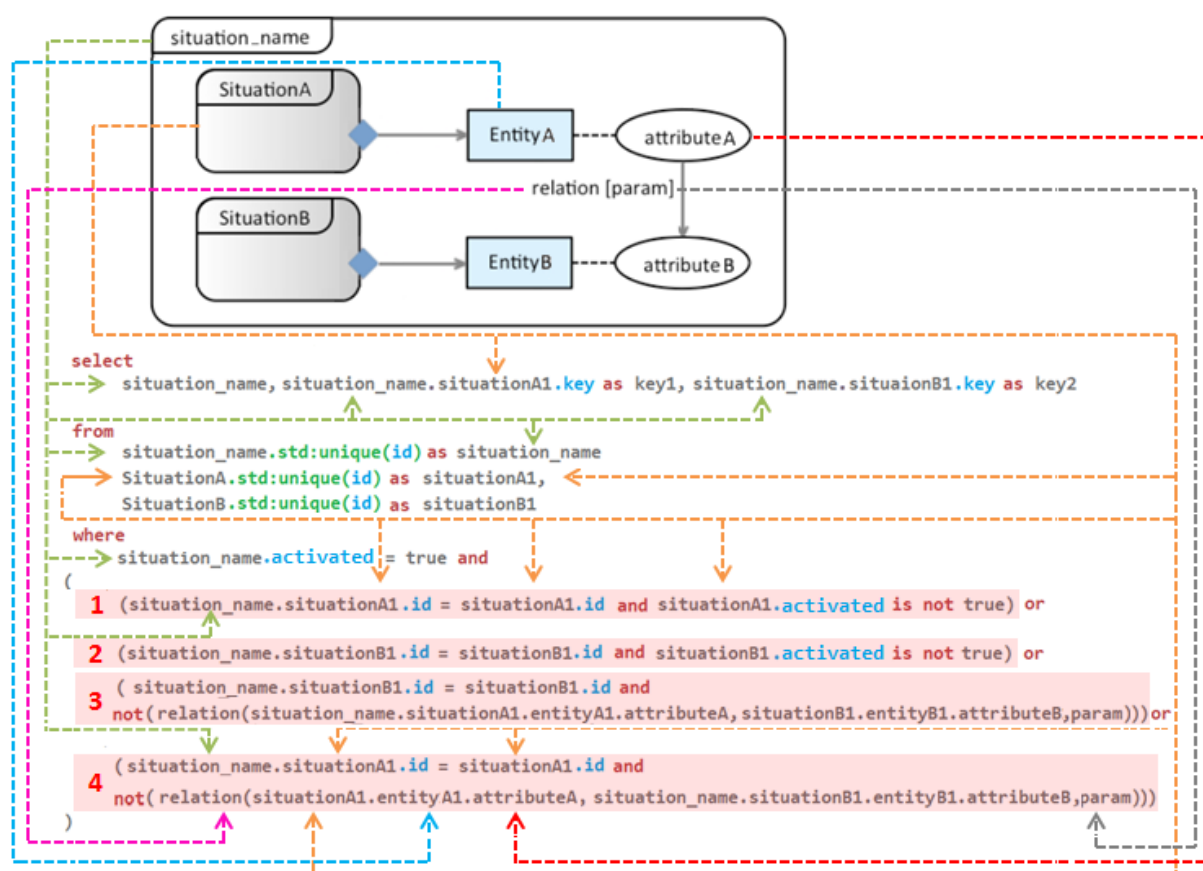


Figura 50. Regra de transformação de desativação do padrão Situação Composta

As principais diferenças encontradas nesse padrão ficam por conta dos trechos 1 e 2 da Figura 50. Como as situações *SituationA* e *SituationB* são situações participantes, além de comparar a referência do participante de mesmo tipo armazenado durante a ativação da situação, é necessário verificar também seus *status* (i.e., se as situações encontram-se ativas ou não). Para representar a negação de uma relação formal envolvendo duas *views* (*SituationA* e *SituationB*, e suas respectivas entidades participantes) é necessário testar todas as possíveis

combinações, de forma análoga ao que foi realizada na cláusula *from* da regra de transformação de ativação do padrão “Situação Composta” (Figura 49). A negação dessa relação é apresentada pelos trechos 3 e 4 da Figura 50.

5.5 CONSIDERAÇÕES DO CAPÍTULO

Nesta seção foram apresentadas as principais regras de transformação utilizadas na geração de regras EPL a partir de modelos SML. Foram ilustradas as regras de transformação responsáveis por cobrir os principais padrões definidos e utilizados por este trabalho. Para cada padrão, foram discutidas as respectivas regras de transformação de ativação e desativação.

No padrão “*Exists*” (seção 5.3), devido às particularidades, foi definido que, ao utilizar uma situação existencial na composição de uma situação, a situação deve possuir apenas um participante. Apesar de ser uma limitação, ainda é possível representar o comportamento desejado, definindo uma nova situação, tendo apenas a situação existencial como participante, e utilizando a nova situação definida como situação participante na definição de uma situação composta.

O capítulo seguinte apresenta um estudo de casos por meio de um cenário de uso. Além disso, é apresentada uma aplicação exemplo, baseada no cenário proposto, a fim de ilustrar a viabilidade de utilizar situações em sistemas baseados em eventos.

6 ESTUDO DE CASOS

Este capítulo tem como objetivo realizar um estudo de caso, a fim de validar a metodologia proposta pelo presente trabalho. Este capítulo está estruturado da seguinte forma: a Seção 6.1 apresenta o cenário utilizado; a Seção 6.2 apresenta as situações modeladas em SML a partir do cenário descrito; a Seção 6.3 apresenta uma aplicação desenvolvida a partir do cenário descrito; a Seção 6.4 apresenta o suporte ferramental utilizado neste trabalho e, por fim; a seção 6.5 apresenta as considerações do capítulo.

6.1 CENÁRIO DE APLICAÇÃO

A fim de exemplificar a metodologia proposta pelo presente trabalho, esta seção tem como objetivo apresentar um cenário de aplicação. O cenário apresentado constitui no mesmo cenário encontrado em (Mielke, 2013), representando um cenário de detecção de fraudes em aplicações bancárias. Neste cenário, usuários acessam suas contas bancárias por meio de dispositivos, classificados em: dispositivos móveis, computadores pessoais ou caixas eletrônicos. A partir das ações do usuário, sendo estas (i) acesso a uma conta (*logged in*) ou; (ii) saque (*withdrawal*), deseja-se detectar comportamentos suspeitos que possam indicar alguma fraude.

Exemplos de padrões de comportamentos suspeitos, apresentados em (Mielke, 2013), são representados por: (i) compras com cartão de crédito com valores anormais ou em categorias de produtos não usuais; (ii) transferência de grandes quantias de dinheiro entre contas não relacionadas; (iii) volume de saques em um dia é maior do que a média de saques durante um mês.

Tomemos como exemplo uma situação em que uma conta é acessada pelo mesmo usuário em duas localizações distintas (ex: 500 km de distância), num curto período de tempo (ex: 10 minutos). Essa situação representa um padrão suspeito, indicando uma possível clonagem de cartão, uma vez que é pouco provável que o

usuário consiga um deslocamento grande num curto período de tempo. Uma vez que tal comportamento seja detectado, ações podem ser automatizadas a fim de fornecer uma melhor e mais ágil solução para o problema. Uma possível ação consiste no envio de uma mensagem ao titular da conta avisando sobre a ação suspeita, de forma que ele possa contactar seu banco caso essa operação não seja de seu conhecimento.

A Figura 51 apresenta o modelo de contexto que descreve as entidades e contextos relevantes para o cenário proposto. Nesse modelo são descritos dois tipos de entidades, dispositivo (*Device*) e conta bancária (*Account*). Um dispositivo (*Device*) é utilizado para acessar uma conta bancária, podendo este ser um computador (*Computer*), dispositivo móvel (*MobileDevice*) ou um caixa eletrônico (*ATM*). Um dispositivo é também uma entidade espacial (*SpatialEntity*), possuindo um contexto intrínseco do tipo localização (*Location*). O contexto relacional de acesso (*Access*) relaciona um dispositivo (*Device*) e uma conta bancária (*Account*), representando a relação existente entre uma conta e o dispositivo pelo qual é acessada. O contexto relacional (*OngoingWithdrawal*) representa a relação existente entre um usuário e um caixa eletrônico durante uma operação de saque, na qual certa quantia de dinheiro é retirada de uma conta. Ambos os conceitos relacionais existem apenas enquanto a relação representada por eles também existe. Por exemplo, o contexto *OngoingWithdrawal* existe apenas enquanto o saque estiver sendo realizado, isto é, desde a requisição até que o dinheiro seja entregue, ou até que a transação seja cancelada.

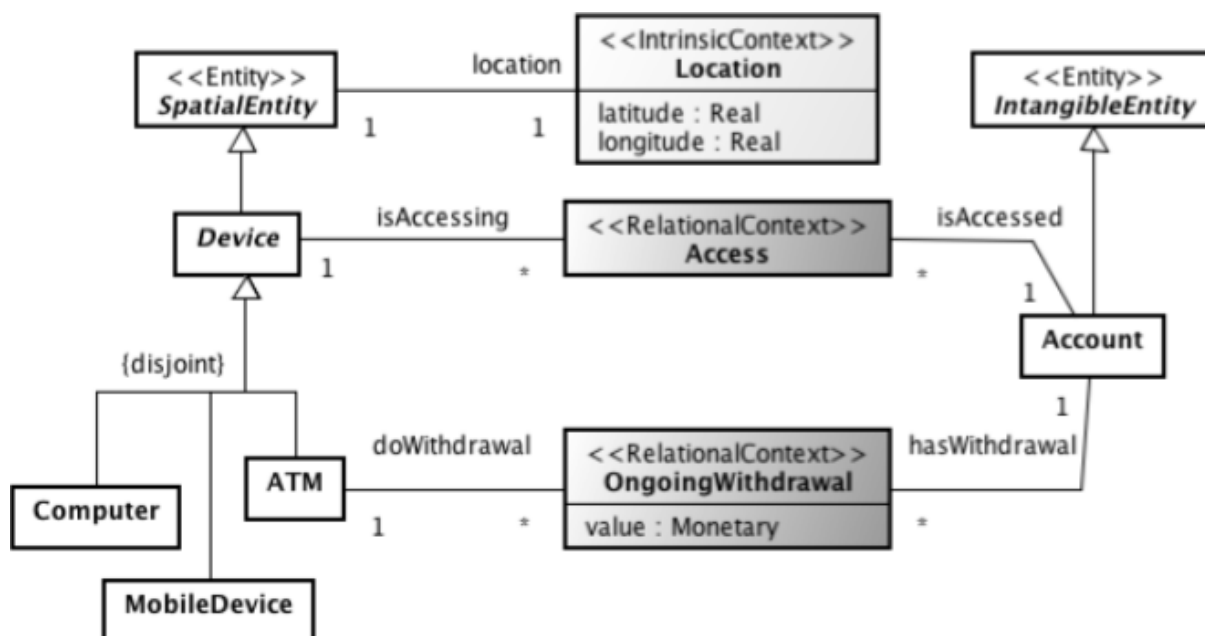


Figura 51. Modelo de contexto para o cenário bancário (Mielke, 2013)

6.2 SITUAÇÕES DESCRITAS PARA O CENÁRIO PROPOSTO

Esta seção apresenta os tipos de situações descritos em (Mielke, 2013) para o cenário apresentado na seção anterior, bem como seus respectivos diagramas SML. Essas situações têm como objetivos detectar comportamentos suspeitos por parte dos usuários, indicando uma possível fraude bancária. Além disso, são apresentados códigos, gerados automaticamente neste trabalho segundo as especificações apresentadas nos capítulos 4 e 5, referente às classes Java representando os tipos de situação descritos nos modelos SML. Por motivos de simplicidade, os respectivos métodos *get* e *set* foram omitidos.

O primeiro tipo de situação, denominada *LoggedIn*, é ilustrada pela Figura 52. Esse é um tipo de situação simples, definido com o intuito de compor demais tipos de situações complexos, não representando por si só um comportamento suspeito. O tipo de situação *LoggedIn* indica que uma conta (*Account*) está sendo acessada (*Access*) por um dispositivo (*Device*).

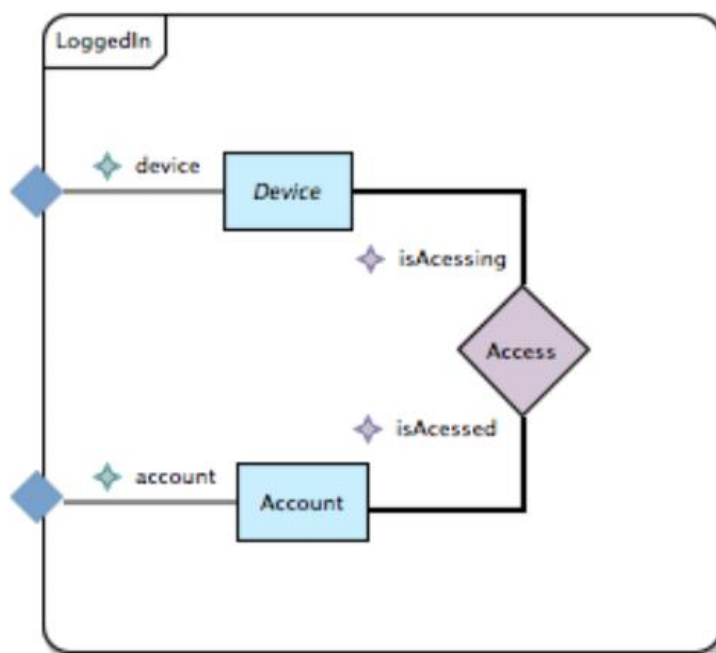


Figura 52. Situação *LoggedIn*

As figuras Figura 53 e Figura 54 apresentam a classe Java, gerada automaticamente, responsável por representar o tipo de situação *LoggedIn* na plataforma SIMPLE. Essa classe contém três atributos dos tipos *Device*, *Account* e *Access* (linhas 3, 4 e 5, Figura 53), representando as entidades contidas no modelo de situação correspondente (Figura 52). Os nomes desses atributos são atribuídos de acordo com os *binding names* especificados no modelo de contexto (*device* e *account*). Uma vez que não foi especificado um *binding name* para a entidade do tipo *Access*, seu nome foi gerado automaticamente, *access1*.

Conforme discutido anteriormente (seção 4.2), toda classe que representa um tipo de situação na plataforma SIMPLE deve estender a classe *Situation* (linha 1, Figura 53) e implementar seu respectivo método construtor (linhas 7 a 31, Figura 53). O método construtor deve inicializar, por meio de seus métodos *set*, os atributos *sitName* (linha 9, Figura 53), *EpIA* (linha 11, Figura 53) e *EpID* (linha 21, Figura 53), contendo respectivamente, o nome do tipo de situação e as regras de ativação e desativação correspondentes. Detalhes sobre as regras de ativação e desativação correspondentes a esse tipo de situação não serão abordados, uma vez que seguem as recomendações e restrições discutidas anteriormente no padrão “Entidade-Contexto Relacional” (seção 5.2).

```

1 public class LoggedIn extends Situation{
2
3     private Device device;
4     private Account account;
5     private Access access1;
6
7     public LoggedIn(){
8
9         setSitName("LoggedIn");
10
11        setEplA("select
12                access1,
13                access1.isAccessing as device,
14                access1.isAccessed as account,
15                access1.key as key1
16            from
17                Access as access1
18            where
19                access1.activated = true");
20
21        setEplD("select LoggedIn, access1.key as key1
22                from
23                    LoggedIn.std:unique(id) as LoggedIn,
24                    Access.std:unique(key) as access1 '
25                where"
26                    LoggedIn.activated = true and
27                    (
28                        LoggedIn.access1.key = access1.key and
29                        access1.activated = false
30                    ) ");
31    }

```

Figura 53. Primeira parte da classe Java correspondente ao tipo de situação *LoggedIn*

A Figura 54 apresenta a segunda parte da classe Java referente ao tipo de situação *LoggedIn*. Essa classe contém a implementação dos métodos *createNewSit* e *doActionAtCreateDeactivationEvent*. Conforme discutido na seção 4.2, esses métodos abstratos, contidos na classe *Situation*, devem ser implementados pelo usuário, representando, respectivamente, os métodos construtores dos eventos de ativação e desativação do tipo de situação correspondente, especificando a estrutura interna de cada um desses eventos.

```

32
33     public Situation createNewSit(EventBean event) {
34         LoggedIn loggedIn = new LoggedIn();
35         loggedIn.setAccount((Account)event.get("account"));
36         loggedIn.setDevice((Device)event.get("device"));
37         loggedIn.setAccess1((Access)event.get("access1"));
38         return loggedIn;
39     }
40
41     public Object doActionAtCreateDeactivationEvent() {
42         LoggedIn loggedIn = new LoggedIn();
43         loggedIn.setAccount(this.getAccount());
44         loggedIn.setDevice(this.getDevice());
45         loggedIn.setAccess1(this.getAccess1());
46         return loggedIn;
47     }
48 }

```

Figura 54. Segunda parte da classe Java correspondente ao tipo de situação *LoggedIn*

O tipo de situação simples *OngoingSuspiciousWithdrawal*, ilustrado pela Figura 55, representa um saque suspeito identificado em uma conta. Os tipos de entidades participantes correspondem ao caixa eletrônico (*ATM*) e a conta (*Account*), ambos relacionados por meio de um contexto relacional de saque (*OngoingWithdrawal*). Um saque suspeito é caracterizado por uma restrição aplicada sobre o valor do saque (*value*), devendo este ser maior que (*greater than*) um literal de valor "1000".

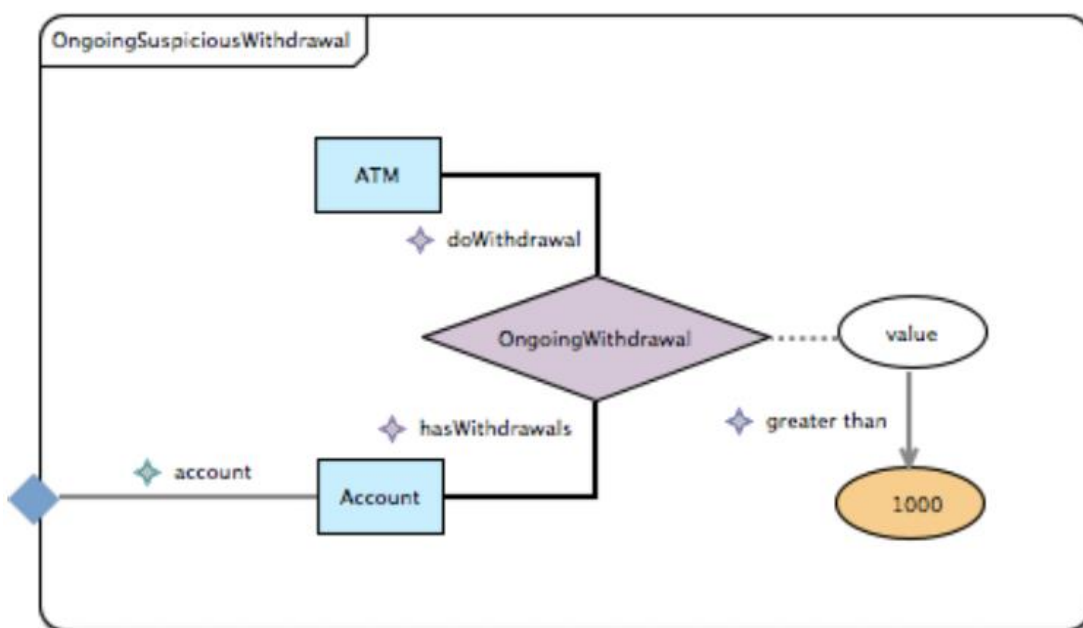


Figura 55. Situação *OngoingSuspiciousWithdrawal*

A Figura 56 apresenta as regras de ativação (linha 1, Figura 56) e desativação (linha 13, Figura 56), geradas automaticamente, referentes ao tipo de situação *OngoingSuspiciousWithdrawal*. A classe Java responsável por representar esse tipo de situação, bem como as regras de ativação e desativação correspondentes, é análogas às apresentadas para o tipo de situação *LoggedIn* (Figura 53 e Figura 54), uma vez que ambos os tipos de situação correspondem ao padrão “Entidade-Contexto Relacional” (seção 5.2). Para prover uma melhor visibilidade e legibilidade das regras de ativação e desativação, as ocorrências do termo *OngoingSuspiciousWithdrawal* foram substituídas pela sigla *OSW*.

```

1  setEplA(
2      "select
3          ongoingWithdrawal1,
4          ongoingWithdrawal1.key as key1,
5          ongoingWithdrawal1.doWithdrawal as aTM1,
6          ongoingWithdrawal1.hasWithdrawal as account
7      from
8          OngoingWithdrawal as ongoingWithdrawal1'
9      where
10         ongoingWithdrawal1.activated = true and
11         ongoingWithdrawal1.value > 1000");
12
13 setEplD(
14     "select OSW, OngoingWithdrawal.key as key1
15     from
16         OSW.std:unique(id) as OSW,
17         OngoingWithdrawal.std:unique(key) as ongoingWithdrawal1
18
19     where
20         OSW.activated = true and
21         (
22             ( OSW.ongoingWithdrawal1.key = ongoingWithdrawal1.key
23               and ongoingWithdrawal1.activated = false )
24         or
25             ( OSW.ongoingWithdrawal1.key = ongoingWithdrawal1.key
26               and not (ongoingWithdrawal1.value > 1000) )
27         )" );

```

Figura 56. Regras de ativação e desativação geradas automaticamente para o tipo de situação *OngoingSuspiciousWithdrawal*

Ao contrário dos tipos de situação *LoggedIn* e *OngoingSuspiciousWithdrawal*, o tipo de situação *SuspiciousParallelLogin* representa um tipo de situação composto. Esse tipo de situação, ilustrado pela Figura 57, representa a ocorrência de acessos

simultâneos a uma conta, sendo definido quando duas ocorrências do tipo de situação *LoggedIn* encontram-se ativas num mesmo instante do tempo.

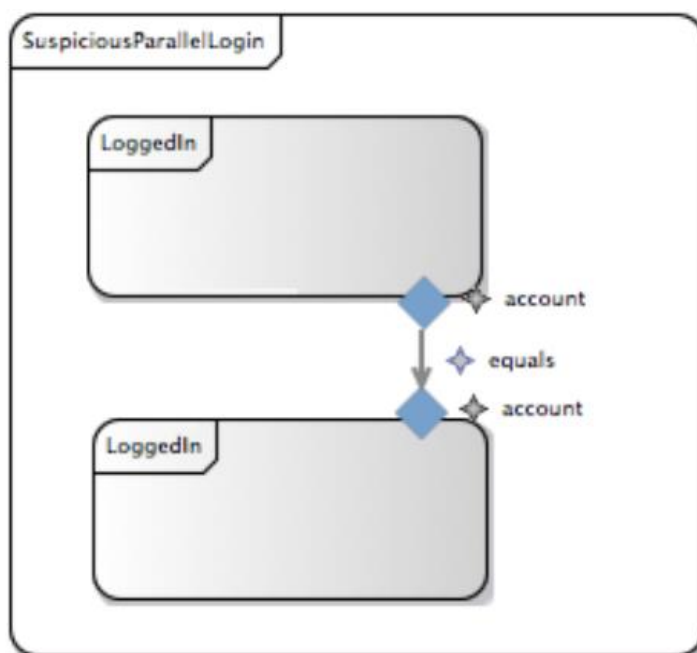


Figura 57. Situação *SuspiciousParallelLogin*

A Figura 58 ilustra o exemplo de uma ocorrência no tempo do tipo de situação *SuspiciousParallelLogin*, baseada em duas ocorrências do tipo de situação *LoggedIn* (para a mesma conta). É importante observar que a situação *SuspiciousParallelLogin* existe apenas durante o intervalo de tempo em que ambas as ocorrências da situação *LoggedIn* existem simultaneamente.

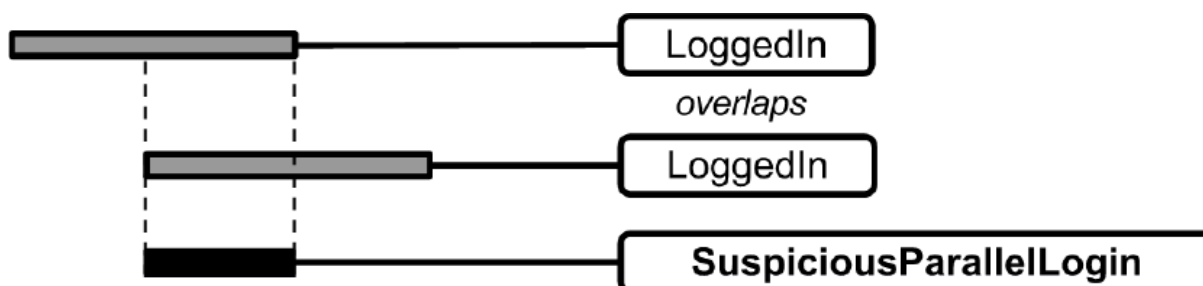


Figura 58. Exemplo de ocorrência no tempo para a situação *SuspiciousParallelLogin*

As figuras Figura 59 e Figura 60 apresentam a classe Java, gerada automaticamente, responsável por representar o tipo de situação *SuspiciousParallelLogin* na plataforma SIMPLE. Essa classe contém dois atributos do tipo de situação *LoggedIn* (linhas 2, Figura 59), representando as entidades

contidas no modelo de situação correspondente (Figura 58). Uma vez que não foi especificado um *binding name* para essas entidades, foi gerado um nome automático para elas, *loggedIn1* e *loggedIn2*.

```

1 public class SuspiciousParallelLogin extends Situation {
2     private LoggedIn loggedIn1, loggedIn2;
3
4     public SuspiciousParallelLogin(){
5         setSitName("SuspiciousParallelLogin");
6         setEpIA('
7             "select
8                 loggedIn1, loggedIn2, loggedIn1.key as key1, loggedIn2.key as key2
9             from
10                 pattern[ every (
11                     ( every loggedIn1 = LoggedIn(activated = true) ->
12                     ( every loggedIn2 = LoggedIn(activated = true) and
13                       not LoggedIn(activated = false, id = loggedIn1.id) ) )
14                   or
15                     ( every loggedIn2 = LoggedIn(activated = true) ->
16                     ( every loggedIn1 = LoggedIn(activated = true) and
17                       not LoggedIn(activated = false, id = loggedIn2.id) ) )
18                 ) ]
19             where
20                 loggedIn1.account = loggedIn2.account" );
21         setEpID(
22             "select SPL,
23                 SPL.loggedIn1.key as key1,
24                 SPL.loggedIn2.key as key2
25             from
26                 SPL.std:unique(id) as SPL,
27                 LoggedIn.std:lastevent() as loggedIn1
28             where
29                 SPL.activated = true
30             (
31                 ( SPL.loggedIn1.id = loggedIn1.id and loggedIn1.activated= false) or
32                 ( SPL.loggedIn2.id = loggedIn1.id and loggedIn1.activated= false) or
33                 ( SPL.loggedIn1.id = loggedIn1.id and
34                   not(loggedIn1.account = SPL.loggedIn2.account) ) or
35                 ( SPL.loggedIn2.id = loggedIn1.id and
36                   not(SPL.loggedIn1.account = loggedIn1.account) )
37             )" );
38     }

```

Figura 59. Primeira parte da classe Java correspondente ao tipo de situação
SuspiciousParallelLogin

Conforme discutido anteriormente, essa classe estende a classe *Situation* (linha 1, Figura 59) e implementa seu método construtor correspondente (linhas 4 a 38, Figura 59), inicializando, por meio de seus métodos *set*, os atributos *sitName* (linha 5, Figura 59), *EpIA* (linha 6, Figura 59) e *EpID* (linha 21, Figura 59), contendo respectivamente, o nome do tipo de situação e as regras de ativação e desativação

correspondentes. Para prover uma melhor visibilidade e legibilidade das regras de ativação e desativação, as ocorrências do termo *SuspiciousParallelLogin* foram substituídas pela sigla *SPL*. Novamente, detalhes sobre as regras de ativação e desativação correspondentes a esse tipo de situação não serão abordados, uma vez que seguem as recomendações e restrições discutidas anteriormente no padrão “Situação Composta” (seção 5.4).

```

39
40     public Situation createNewSit(EventBean event) {
41         SuspiciousParallelLogin suspiciousParallelLogin = new SuspiciousParallelLogin();
42         suspiciousParallelLogin.setLoggedIn1((LoggedIn)event.get("loggedIn1"));
43         suspiciousParallelLogin.setLoggedIn2((LoggedIn)event.get("loggedIn2"));
44         return suspiciousParallelLogin;
45     }
46
47     public Object doActionAtCreateDeactivationEvent() {
48         SuspiciousParallelLogin suspiciousParallelLogin = new SuspiciousParallelLogin();
49         suspiciousParallelLogin.setLoggedIn1(this.getLoggedIn1());
50         suspiciousParallelLogin.setLoggedIn2(this.getLoggedIn2());
51         return suspiciousParallelLogin;
52     }
53 }

```

Figura 60. Segunda parte da classe Java correspondente ao tipo de situação *SuspiciousParallelLogin*

A Figura 61 ilustra o tipo de situação *SuspiciousFarawayLogin*, representando a ocorrência de acessos (situações do tipo *LoggedIn*) à mesma conta realizados em locais geograficamente distantes (pelo menos 500 km de distância), num curto período de tempo (duas horas). É importante observar que a primeira ocorrência do tipo de situação *LoggedIn* representa uma situação passada (fundo branco), enquanto a segunda representa uma situação presente (fundo cinza).

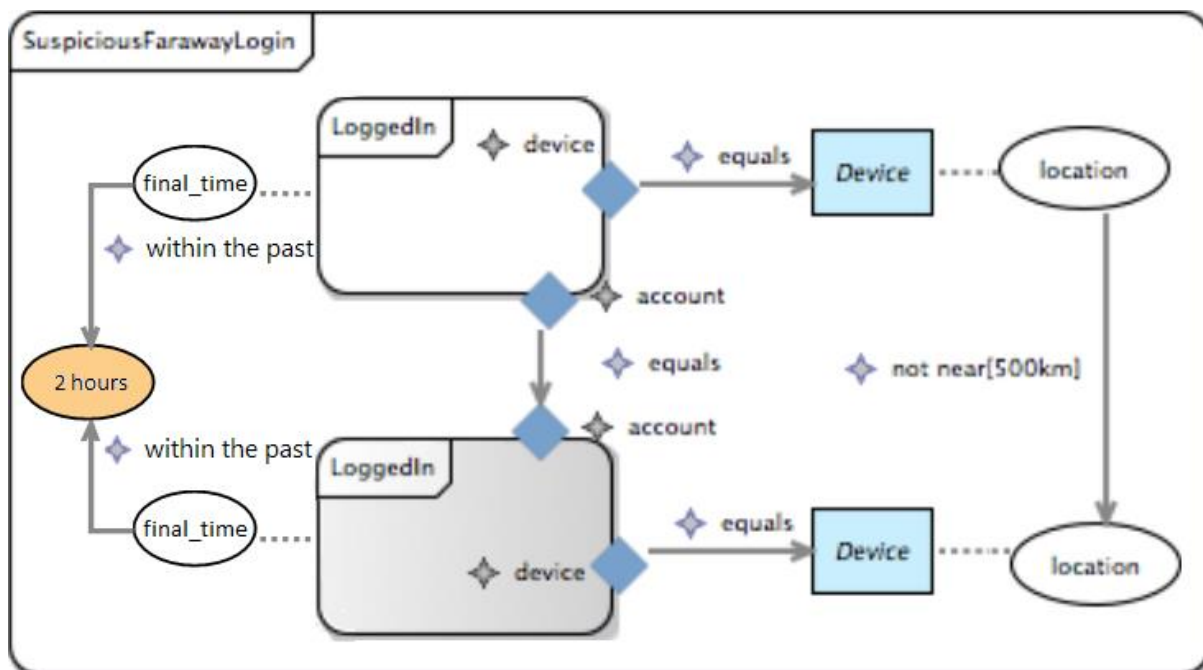


Figura 61. Situação *SuspiciousFarawayLogin*

A Figura 62 ilustra o exemplo de uma ocorrência no tempo do tipo de situação *SuspiciousFarawayLogin*, baseada em duas ocorrências do tipo de situação *LoggedIn* (para a mesma conta). A situação *SuspiciousFarawayLogin* é ativada simultaneamente à ativação da segunda ocorrência da situação do tipo *LoggedIn*, iniciada uma hora após o término da primeira ocorrência. No momento em que a segunda ocorrência da situação do tipo *LoggedIn* é desativada, o mesmo ocorre com a ocorrência da situação do tipo *SuspiciousFarawayLogin*.

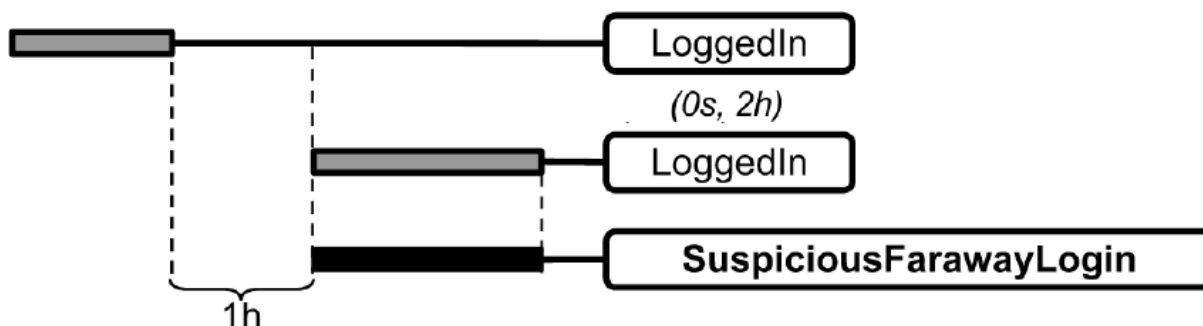


Figura 62. Exemplo de ocorrência no tempo para a situação *SuspiciousFarawayLogin*

A Figura 63 apresenta as regras de ativação (linha 1, Figura 63) e desativação (linha 13, Figura 63), geradas automaticamente, referentes ao tipo de situação *SuspiciousFarawayLogin*. Para prover uma melhor visibilidade e legibilidade das

regras de ativação e desativação, as ocorrências do termo *SuspiciousFarawayLogin* foram substituídas pela sigla *SFL*. A classe Java responsável por representar esse tipo de situação, bem como as regras de ativação e desativação correspondentes, são análogas às apresentadas para o tipo de situação *SuspiciousParallelLogin* (Figura 59 e Figura 60), uma vez que ambos os tipos de situação correspondem ao padrão “Situação Composta” (seção 5.4). A diferença é dada pelo uso de uma janela temporal, indicando que ambas as situações do tipo *LoggedIn* devem ter ocorrido dentro do prazo de duas horas (linhas 7 e 11, Figura 63), e da função *near* negada pela operador *not* (linha 15, Figura 63), indicando que ambos os dispositivos (*Device*) devem estar pelo menos a 500 km de distância um do outro.

```

1  setEplA(
2    " select loggedInIn1, loggedInIn2, loggedInIn1.key as key1, loggedInIn2.key as key2
3    from
4      pattern[ every (
5        ( every loggedInIn1 = LoggedIn(activated = false) ->
6          ( every loggedInIn2 = LoggedIn(activated = true ) and
7            not LoggedIn(activated = true, id = loggedInIn1.id)) where timer:within(2 hours) )
8        or
9        ( every loggedInIn2 = LoggedIn(activated = true ) ->
10         ( every loggedInIn1 = LoggedIn(activated = false) and
11           not LoggedIn(activated = false, id = loggedInIn2.id)) where timer:within(2 hours) )
12      ) ]
13    where
14      loggedInIn1.account = loggedInIn2.account and
15      not(near(loggedIn1.device.location, loggedInIn2.device.location,500))" );
16
17  setEplD(
18    " select SFL, SFL.loggedInIn1.key as key1, SFL.loggedInIn2.key as key2
19    from
20      SuspiciousFarawayLogin.std:unique(id) as SFL,
21      LoggedIn.std:lastevent() as loggedInIn1
22    where
23      SFL.activated = true
24      and
25      (
26        (SFL.loggedInIn1.id = loggedInIn1.id and loggedInIn1.activated = true ) or
27        (SFL.loggedInIn2.id = loggedInIn1.id and loggedInIn1.activated = false) or
28        (SFL.loggedInIn1.id = loggedInIn1.id and not(loggedIn1.account = SFL.loggedInIn2.account)) or
29        (SFL.loggedInIn2.id = loggedInIn1.id and not(SFL.loggedInIn1.account = loggedInIn1.account))
30        (SFL.loggedInIn1.id = loggedInIn1.id and
31          not(not(near(loggedIn1.device.location, SFL.loggedInIn2.device.location,500)))) or
32          (SFL.loggedInIn2.id = loggedInIn1.id and
33            not(not(near(SFL.loggedInIn1.device.location, loggedInIn1.device.location,500))))
34      ) );

```

Figura 63. Regras de ativação e desativação geradas automaticamente para o tipo de situação *SuspiciousFarawayLogin*

A Figura 64 ilustra o tipo de situação *AccountUnderObservation*, representando a observação de uma conta em que são realizados saques suspeitos nos últimos 30

dias. A conta em questão deixa de estar em observação apenas quando não existirem novas ocorrências de saques suspeitos para aquela conta nos últimos 30 dias.

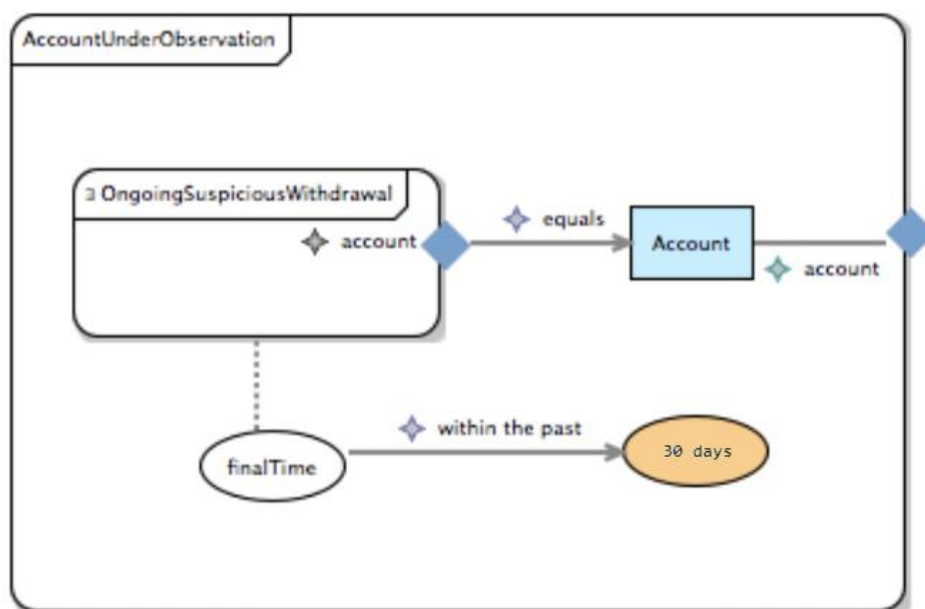


Figura 64. Situação *AccountUnderObservation*

A Figura 65 ilustra o exemplo de uma ocorrência no tempo do tipo de situação *AccountUnderObservation*, baseada em duas ocorrências do tipo de situação *OngoingSuspiciousWithdrawal* (para a mesma conta). É importante observar que a situação do tipo *AccountUnderObservation* é iniciada juntamente à primeira ocorrência da situação do tipo *OngoingSuspiciousWithdrawal*, iniciando uma janela temporal de 30 dias, sendo esta estendida por mais 30 dias a cada nova ocorrência de uma situação do tipo *OngoingSuspiciousWithdrawal*.

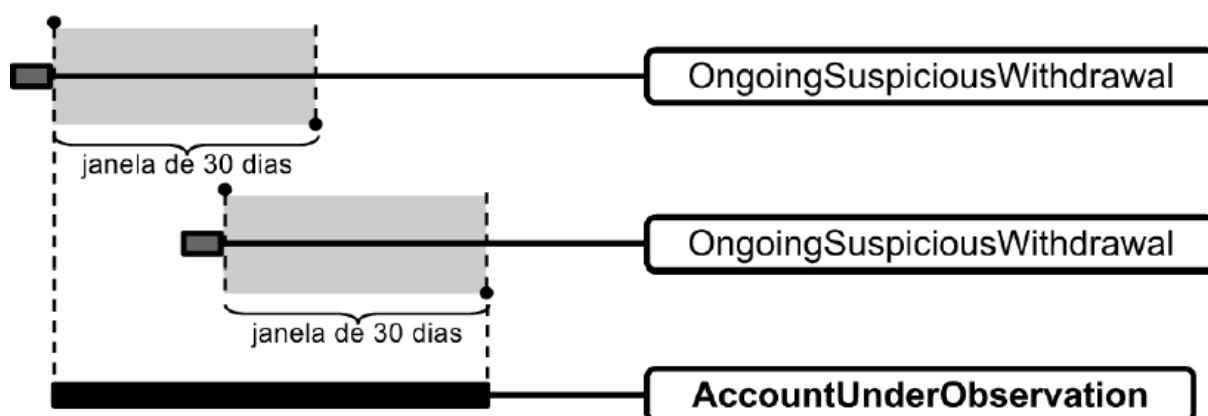


Figura 65. Exemplo de ocorrência no tempo para a situação *AccountUnderObservation*

A Figura 66 apresenta as regras de ativação (linha 1, Figura 66) e desativação (linha 13, Figura 66), referentes ao tipo de situação *AccountUnderObservation*, geradas automaticamente de acordo com as especificações e restrições discutidas a respeito do padrão “*Exists*” (seção 5.3).

```

1  setEplA(
2      " select
3          OngoingSuspiciousWithdrawal,
4          OngoingSuspiciousWithdrawal.account as account
5      from
6          OngoingSuspiciousWithdrawal as OngoingSuspiciousWithdrawal
7      where
8          OngoingSuspiciousWithdrawal.activated = false");
9
10 setEplD(
11     " select AccountUnderObservation
12     from
13         pattern[
14             AccountUnderObservation = AccountUnderObservation ->
15             every (
16                 timer:interval( 30 days ) and
17                 not OngoingSuspiciousWithdrawal(activated = false)
18             )
19         ] ");

```

Figura 66. Regras de ativação e desativação geradas automaticamente para o tipo de situação *AccountUnderObservation*

6.3 APLICAÇÃO EXEMPLO

Esta seção apresenta uma aplicação executável para o cenário apresentado na seção 6.1. Com isso, visa-se demonstrar a viabilidade da utilização de situações, através da plataforma SIMPLE, em cenários reais. A aplicação desenvolvida é dividida em duas partes: (i) a primeira simulando um cliente, permitindo ao usuário acessar sua conta e simular saques e; (ii) a segunda simulando uma central de monitoramento bancário, possibilitando verificar a ativação e desativação das situações.

A Figura 67 ilustra a aplicação responsável por simular o cliente. Nessa aplicação o usuário informa sua conta (*Account*), seleciona o tipo de dispositivo

(*Device*) que deseja simular (*ATM*, *MobileDevice* ou *Computer*), além de escolher uma localização (*Location*) a fim de simular situações que envolvam diferença de distâncias, como é o caso da situação *SuspiciousFarawayLogin*. Após preencher os dados, o cliente deve acessar a sua conta através do botão “Access”, criando um contexto relacional do tipo *Access*. A criação desse contexto relacional tem como consequência a criação de uma instância (representando sua ativação) do tipo de situação *LoggedIn*, associada a conta especificada pelo usuário.

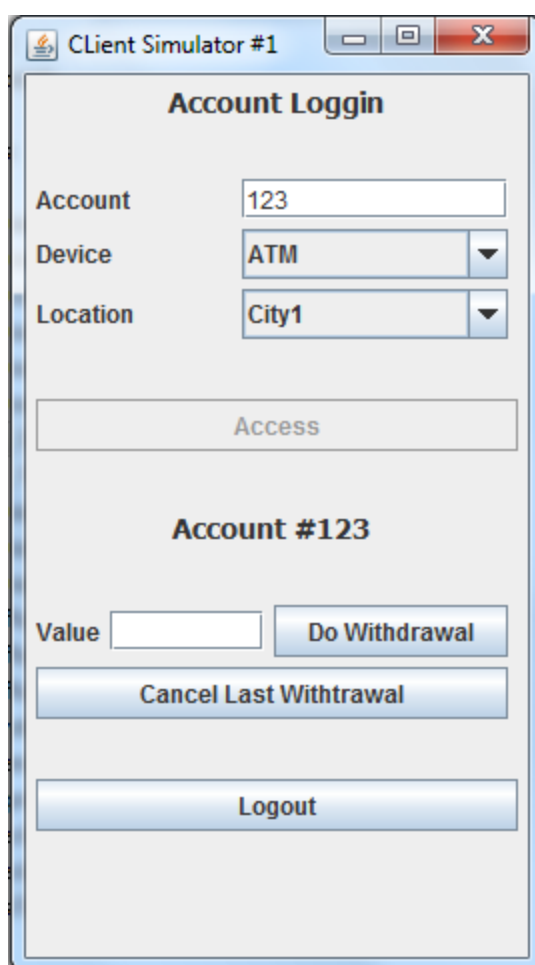
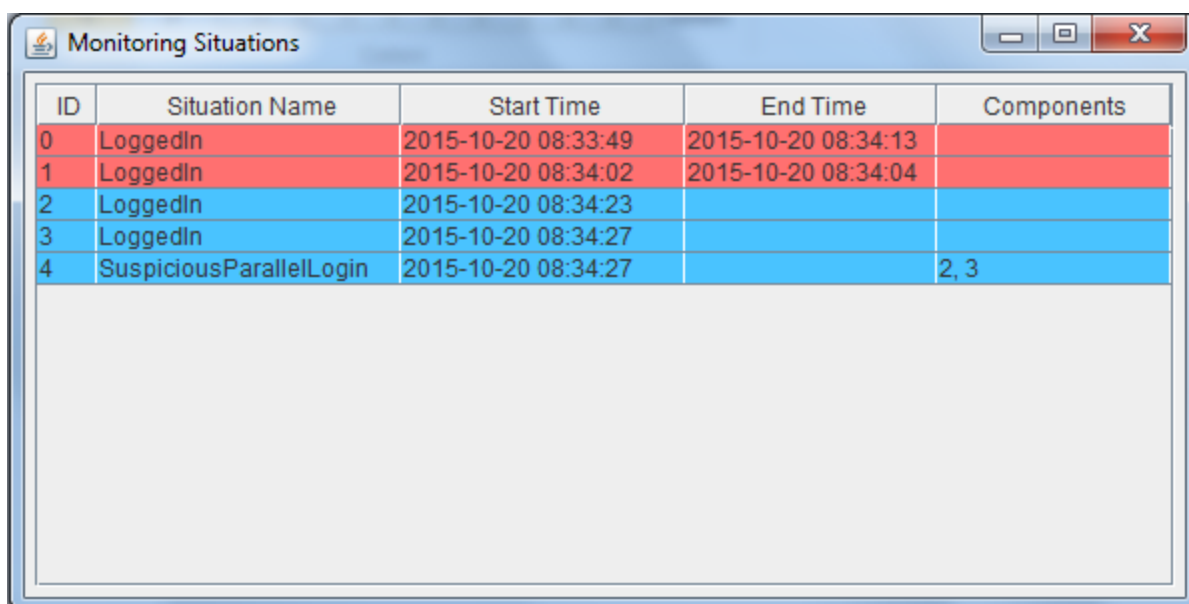


Figura 67. Aplicação simulando um cliente

Caso o dispositivo selecionado seja do tipo *ATM*, é possível que o cliente realize um saque, através do botão “*Do Withdrawal*”. Esse botão cria um contexto relacional do tipo *OngoingWithdrawal*, com seu valor igual ao especificado no campo “*Value*”. A criação desse contexto relacional tem como consequência a criação de uma instância (representando sua ativação) do tipo de situação *OngoingSuspiciousWithdrawal*, associada a conta especificada pelo usuário. Além

disso, é possível que o usuário cancele sua operação de saque, através do botão “*Cancel Last Withdrawal*”, levando à exclusão do contexto relacional criado anteriormente. Como consequência, a instância correspondente do tipo de situação *OngoingSuspiciousWithdrawal* é desativada.

A ativação de situações simples do tipo *LoggedIn* e *OngoingSuspiciousWithdrawal*, leva, eventualmente, a ativação de situações de tipos mais complexas, como os tipos de situação *SuspiciousParallelLogin*, *SuspiciousFarawayLogin* e *AccountUnderObservation*. As ativações e desativações dessas situações são observadas através da aplicação que simula uma central de monitoramento bancário, ilustrada pela Figura 68. Nessa figura são exibidas informações relevantes sobre as instâncias das situações, exibidas na tela de monitoramento à medida que transações são realizadas pelos usuários.



ID	Situation Name	Start Time	End Time	Components
0	LoggedIn	2015-10-20 08:33:49	2015-10-20 08:34:13	
1	LoggedIn	2015-10-20 08:34:02	2015-10-20 08:34:04	
2	LoggedIn	2015-10-20 08:34:23		
3	LoggedIn	2015-10-20 08:34:27		
4	SuspiciousParallelLogin	2015-10-20 08:34:27		2, 3

Figura 68. Aplicação simulando uma central de monitoramento bancário

O campo “*ID*” apresenta o identificador único atribuído a cada instância de um tipo de situação. O campo “*Situation Name*” apresenta o tipo de situação instanciado pela situação correspondente. Os campos “*Start Time*” e “*End Time*” representam, respectivamente, os tempos de ativação e desativação de cada situação. Situações coloridas na cor azul indicam situações ativas e, por esse motivo, não apresentam um valor para o campo “*End Time*”. Uma vez que a situação seja desativada, a mesma é colorida na cor vermelha e tem o valor do tempo atual associado ao seu

campo “*End Time*”. Por fim, o campo “*Components*” é utilizado apenas para situações compostas. Esse campo informa o ID das instâncias de situações mais simples que levaram a ativação da instância correspondente. Por exemplo, na Figura 68, é possível observar que a situação *SuspiciousParallelLogin* (ID = 4) foi ativada a partir da ativação de duas instâncias do tipo de situação *LoggedIn*, cujos valores de campo ID são iguais a 2 e 3.

Possíveis ações a serem tomadas a partir da detecção das situações não são suportas pela aplicação atual, uma vez que fogem do escopo do trabalho atual. Para especificar tais ações, cabe ao usuário implementar os *Listeners* correspondentes a cada tipo de situação na plataforma SIMPLE.

6.4 SUPORTE FERRAMENTAL

Esta seção tem como objetivo apresentar as ferramentas utilizadas pelo presente trabalho para apoiar o uso de uma abordagem de desenvolvimento orientada a modelos. As seções seguintes apresentam as duas ferramentas utilizadas: (i) *Obeo Designer* e (ii) *Acceleo*.

6.4.1 Obeo Designer

A ferramenta *Obeo Designer*⁷ permite o desenvolvimento e uso de editores gráficos de modelos para qualquer domínio, integrados a IDE Eclipse⁸. *Obeo Designer* é baseada em GMF⁹ (*Graphical Modeling Framework*) e EMF¹⁰ (*Eclipse Modeling Framework*), utilizando a linguagem da ferramenta *Acceleo* (seção 6.4.2) para implementar algumas funcionalidades.

Essa ferramenta pode ser dividida em duas partes: *tooling* e *runtime*. *Tooling* corresponde aos componentes voltados à utilização por parte dos desenvolvedores,

⁷ <http://www.obeodesigner.com/>

⁸ <https://eclipse.org/>

⁹ <http://www.eclipse.org/modeling/gmp/>

¹⁰ <http://www.eclipse.org/modeling/emf/>

provendo um ambiente rico para a especificação de modelos, manipulados pelo usuário final de forma gráfica. Essa etapa é realizada de forma declarativa através da configuração de um *description model*. Cada modelo deve conter um *description model* correspondente, onde o desenvolvedor deve especificar:

1. Quais elementos do modelo do domínio são visíveis ou não;
2. Informações visuais a respeito dos elementos;
3. O comportamento dos elementos: como a ferramenta e os usuários devem interagir com cada elemento;

O resultado final da etapa de *tooling* consiste num conjunto de *description models*, algumas vezes acompanhado de algum código Java, podendo os mesmos ser disponibilizados para o usuário na forma de um *plug-in Eclipse*.

Por sua vez, *runtime*, corresponde aos componentes voltados à utilização por parte dos usuários finais, responsável por interpretar os *description models* especificados na etapa de *tooling* e apresentar tais modelos ao usuário de acordo com o visual e comportamento especificados. Isso é realizado através *representation models*, nos quais descrevem e apresentam ao usuário final uma representação concreta referente aos *description models*. Os *representation models* são apresentados ao usuário através de um editor gráfico, permitindo ao usuário interagir sobre o mesmo tomando como base as descrições e restrições impostas na etapa de *tooling*. A etapa de *runtime* permite que o usuário final crie e edite novos modelos sem a necessidade de qualquer conhecimento a respeito da etapa de *tooling*.

Uma vez que a linguagem SML foi construída com base na etapa de *tooling*, o presente trabalho apenas utilizou a ferramenta *Obeo Designer* para a etapa de *runtime*, gerando uma representação concreta, ou seja, uma ferramenta gráfica, dos modelos e metamodelos especificados pela linguagem SML. Com a utilização de tal ferramenta, foi possível especificar os tipos de situações de forma gráfica, tornando o processo mais fácil e rápido, permitindo sua utilização num nível maior de abstração.

Atualmente, a ferramenta *Obeo Designer* foi descontinuada, sendo substituída pela ferramenta *Sirius*¹¹. *Sirius* mantém os mesmos recursos de seu antecessor, porém seu uso é gratuito, ao contrário do *Obeo Designer* que necessita de uma licença de uso. Como SML foi desenvolvida utilizando a ferramenta *Obeo Designer*, o presente trabalho fez uso da mesma ferramenta (e na mesma versão) a fim de garantir uma correta representação concreta dos *descriptions models* descritos pela linguagem SML, evitando qualquer tipo de problema originado por falta de compatibilidade com versões anteriores.

6.4.2 Acceleo

Acceleo é uma ferramenta de transformação de modelo para texto, segundo a especificação de *Model to Text Language* (MTL) da OMG (OMG, 2012). *Acceleo* necessita de um metamodelo e de um modelo (compatível com o metamodelo), gerando automaticamente texto a partir do modelo em questão. Essa ferramenta é disponibilizada, assim como o *Obeo Designer*, na forma de um *plug-in Eclipse*, permitindo uma fácil instalação e integração com a IDE *Eclipse*.

O metamodelo e o modelo utilizados são definidos utilizando EMF (*Eclipse Modeling Framework*), tornando *Acceleo* compatível com outras ferramentas baseadas em EMF, em particular, com a ferramenta *Obeo Designer*. Dessa forma, modelos especificados pela ferramenta gráfica do *Obeo Designer* podem ser convertidos para código utilizando a ferramenta *Acceleo* de forma direta. No presente trabalho, os modelos SML descritos utilizando a *runtime* do *Obeo Designer* foram utilizados como entrada pela plataforma *Acceleo*, com o intuito de gerar de forma automática código Java correspondente para ser executado diretamente na plataforma SIMPLE.

A transformação de modelo para código é realizada por meio de *templates*, nos quais implementam regras de transformação. Estas, por sua vez, mapeiam elementos do metamodelo (e logo, do modelo) para o texto que deve ser gerado, de forma direta, refletindo a estrutura do *template* diretamente na estrutura do texto.

¹¹ <http://www.obeodesigner.com/sirius>

Além disso, *Acceleo* possui uma linguagem que pode ser utilizada juntamente com os *templates*, de forma a permitir estruturas condicionais, laços e variáveis (dentro de um dado escopo).

Apesar de a ferramenta *Acceleo* ter facilitado a geração automática de código, foram encontrados alguns problemas, como a ausência de variáveis globais, ausências de funções e, principalmente, por não permitir a definição de estruturas de dados mais complexas, por exemplo, listas ou tabelas *hash*. Isso se tornou uma limitação durante a elaboração do presente trabalho. Para contornar tais limitações, sempre que era necessário refinar um conjunto de dados, de forma a extrair informações relevantes, foi necessário realizar repetidas vezes os mesmos processamentos, uma vez que não existem estruturas capazes de armazenar os resultados obtidos previamente.

6.5 CONSIDERAÇÕES DO CAPÍTULO

Este capítulo apresentou um estudo de casos com base no cenário de aplicação apresentado em (Mielke, 2013). Baseado nos modelos de contexto e situação apresentados em (Mielke, 2013), foi gerado, de forma automática, código que represente as entidades e tipos de situações modelados. Esse código, gerado com o auxílio das ferramentas *Obeo Designer* e *Acceleo*, é executado diretamente sobre a plataforma SIMPLE, de forma que tal plataforma seja capaz de processar, detectar e gerenciar o ciclo de vida das situações modeladas previamente. Com base nesse código, foi apresentada uma aplicação baseada no cenário apresentado, com o intuito de ilustrar a viabilidade da utilização de situações em sistemas baseados em eventos num cenário real.

O capítulo seguinte apresenta uma avaliação de desempenho desenvolvida com o intuito de mensurar o impacto da utilização da plataforma SIMPLE sobre a plataforma Esper. É apresentada a metodologia proposta para a elaboração e realização dos testes, e, uma vez apresentados os resultados obtidos, os mesmos são discutidos e avaliados.

7 AVALIAÇÃO DE DESEMPENHO

Conforme discutido anteriormente na seção 2.2, sistemas de CEP possuem fortes requisitos associados, como (i) escalabilidade, (ii) grande capacidade de processamento, (iii) resposta em tempo real e (iv) disponibilidade. Um dos objetivos do presente trabalho consistiu na implementação da plataforma SIMPLE, com o intuito de permitir que uma plataforma Esper seja capaz de detectar e processar situações adequadamente. A introdução de uma nova camada de abstração acarreta, naturalmente, certo impacto sobre a plataforma utilizada como alvo, por exemplo, um aumento no tempo de resposta, uma vez que são acrescentadas novas funcionalidades e, conseqüentemente, um maior tempo para processamento. Dessa forma, faz-se necessário a realização de uma análise de desempenho com o objetivo de mensurar o impacto sobre a plataforma Esper, proveniente da utilização da plataforma SIMPLE.

Este capítulo está organizado da seguinte forma: a Seção 7.1 apresenta a metodologia utilizada para a elaboração dos testes de desempenho; a Seção 7.2 apresenta e discute os resultados obtidos por meio dos testes executados e, por fim, a Seção 7.3 apresenta as considerações do capítulo.

7.1 METODOLOGIA

Esta seção tem como objetivo apresentar e discutir a metodologia utilizada para realização e avaliação dos testes, bem como ferramentas utilizadas para tal finalidade. A seção 7.1.1 apresenta as configurações de hardware utilizadas nos experimentos, enquanto a seção 7.1.2 apresenta detalhes sobre a elaboração e execução dos experimentos.

7.1.1 Configuração

Devido a limitações técnicas encontradas em nosso ambiente físico de testes, não foi possível realizar os experimentos de forma distribuída. Dessa forma, todos os experimentos aqui descritos foram realizados de forma local, sempre utilizando o mesmo computador e apenas um computador. A Figura 69 apresenta as configurações de *hardware* do computador utilizado nos experimentos.

Sistema Operacional :	Windows 7 Professional - Service Pack 1
Tipo de sistema:	Sistema Operacional de 64 Bits
Fabricante:	Dell
Modelo:	Optiplex 7010
Processador:	Intel(R) Core(TM) i5-3570 CPU @ 3.40GHz 3.40 GHz
Memória instalada (RAM):	4,00 GB

Figura 69. Configuração do computador utilizado ao longo dos experimentos

Uma vez que a plataforma Esper encontra-se disponível na linguagem Java, a mesma tem seu desempenho diretamente afetado pelas configurações da *Java Virtual Machine* (JVM). Nesse caso, também foi utilizada uma configuração de *software* específica, referente à JVM. A partir das recomendações apresentadas na seção de desempenho do manual de uso da plataforma Esper e de testes empíricos realizados durante o desenvolvimento deste trabalho, foram utilizadas em todos os testes as configurações apresentadas pela Figura 70. Dado que não existe precisão quanto ao melhor conjunto de parâmetros para cada ambiente, e que os mesmos foram obtidos de forma empírica, é possível que outras configurações atinjam níveis de desempenho melhor do que os encontrados pelo presente trabalho.

```
-Xms2g -Xmx2g -XX:NewSize=128m -XX:MaxNewSize=128m
```

Figura 70. Configurações utilizadas na JVM ao longo dos experimentos

7.1.2 Experimentos

Os experimentos realizados pelo presente trabalho foram executados com base no *Esper Benchmark Kit*¹². Esse módulo, disponibilizado juntamente com a plataforma Esper, tem como objetivo permitir ao usuário mensurar o desempenho da plataforma Esper. O *Esper Benchmark Kit* consiste, basicamente, numa aplicação servidor, desenvolvida em cima da plataforma Esper, responsável por escutar clientes remotos por meio do protocolo TCP. Clientes enviam remotamente ao servidor dados do tipo *MarketData* (representando um produto simples), definindo três atributos: (i) *ticker*, um campo do tipo *String* representando uma *tag* responsável por identificar cada produto; (ii) *price*, um campo do tipo *double* indicando o valor do produto e; (iii) *volume*, um campo do tipo *int* indicando um volume para cada produto. Diversos parâmetros são disponibilizados, a fim de possibilitar variações na configuração de clientes e servidor, como especificar o número de eventos enviados a cada teste ou a taxa de eventos enviados por segundo.

Uma vez que os experimentos foram realizados em apenas um computador, todos os clientes foram simulados localmente, utilizando-se recursos disponibilizados pelo próprio *Esper Benchmark Kit* para tal finalidade. A Figura 71 ilustra uma possível saída obtida a partir da utilização do *Esper Benchmark Kit*.

O *Esper Benchmark Kit*, fornece estatísticas a respeito da latência e vazão obtidos pela máquina Esper durante cada teste. A latência, medida em nanosegundos (ns), indica o tempo médio gasto pela plataforma para processar cada evento, tendo como tempo inicial o seu tempo de criação. A vazão, medida em eventos por segundo (evt/s), indica a quantidade média de eventos processados pela plataforma por unidade de tempo.

¹² <http://www.espertech.com/esper/performance.php>

```

---Stats - engine (unit: ns)

```

1	Avg:	2528	#4101107			
2	0 <	5000:	97.01%	97.01%	#3978672	
3	5000 <	10000:	2.60%	99.62%	#106669	
	10000 <	15000:	0.35%	99.97%	#14337	
	15000 <	20000:	0.02%	99.99%	#971	
	20000 <	25000:	0.00%	99.99%	#177	
	25000 <	50000:	0.00%	100.00%	#89	
	50000 <	100000:	0.00%	100.00%	#41	
	100000 <	500000:	0.00%	100.00%	#120	
	500000 <	1000000:	0.00%	100.00%	#2	
	1000000 <	2500000:	0.00%	100.00%	#7	
	2500000 <	5000000:	0.00%	100.00%	#5	
	5000000 <	more:	0.00%	100.00%	#18	
	Throughput	412503	(active 0	pending 0	cnx 4)	4

Figura 71. Exemplo de saída utilizando o *Esper Benchmark Kit*

Para melhor entendimento dos dados apresentados na saída ilustrada pela Figura 71, a mesma foi dividida em quatro partes enumeradas. A primeira linha enumerada indica que o teste apresentou uma latência média correspondente a 2528 ns, considerando o tempo de processamento medido para um total de 4.101.107 eventos. As demais linhas, exceto a última, indicam intervalos de latência. Por exemplo, a segunda linha indica que para o teste um total de 3.978.672 eventos obtiveram sua respectiva latência entre 0 e 5us, representando 97.01% dos eventos totais processados. A terceira linha, de forma análoga, indica que um total de 106.669 eventos alcançaram uma latência entre 5 e 10us, representando 2.60% dos eventos totais processados. Com isso, um total acumulado de 4.085.341 eventos (equivalente à soma dos valores 3.978.672 e 106.669) alcançaram uma latência menor que 10us, representando 99,62% dos eventos totais. A última indica que a vazão média alcançada durante o teste foi de 412.503 eventos/s. Além disso, é exibido que o teste fez uso de 4 clientes conectados (*cnx*), sem a ocorrência de clientes pendentes (*pending*) ou ativos (*active*) ao final do teste.

Tendo como base os dados apresentados como saída no *Esper Benchmark Kit*, o presente trabalho dividiu a etapa de teste em duas. A primeira etapa consistiu em executar o *Esper Benchmark Kit* apenas utilizando a plataforma Esper. Nessa etapa, foram executadas 5 baterias de testes. Cada bateria de teste consistiu em 56 testes sequenciais, com duração de 5 segundos cada teste, no qual clientes locais enviam eventos (objetos) do tipo *MarketData*, a uma taxa de envio de 100.000 eventos/s, a um servidor onde encontra-se a plataforma Esper. Cada evento enviado teve o valor de seu atributo *ticker* atribuído de forma sequencial (considerando o valor máximo para o atributo *ticker* como 1000). Além disso, foi registrada a *query* “*select * from Market(ticker='\$')*” juntamente ao servidor Esper, de forma que fosse registrada uma *query* para cada valor do atributo *ticker* distinto. À medida que uma nova bateria era executada, a quantidade de clientes utilizados era modificada. Cada uma das cinco baterias utilizou um cliente, dois clientes, três clientes, quatro clientes e dez clientes, respectivamente. Todas as cinco baterias de testes foram repetidas três vezes, de forma a verificar que os resultados médios eram mantidos e garantir que tais resultados não foram afetados por interferências externas. A Figura 72 apresenta uma tabela contendo o resumo das baterias de testes realizadas.

Nº da Bateria	Nº de Clientes	Ticker máximo	Query	Repetições
1	1	1000	<i>select * from Market(ticker='\$')</i>	3
2	2			
3	3			
4	4			
5	10			

Figura 72. Tabela contendo o resumo das baterias de testes realizadas

A segunda etapa de testes, por sua vez, consistiu em executar novamente as cinco baterias de testes anteriores (cada uma sendo repetida três vezes), porém, utilizando o *Esper Benchmark Kit* juntamente com a plataforma Esper e a plataforma SIMPLE. Nessa etapa, além da *query* registrada anteriormente, foram acrescentados dois tipos de situações, *Sit1* e *Sit2*, sendo responsabilidade da plataforma SIMPLE gerenciar e detectar o ciclo de vida das mesmas. A Figura 73 apresenta as regras de ativação e desativação correspondentes aos tipos de situação *Sit1* e *Sit2*. As regras apresentadas na Figura 73 foram especificadas

segundo os padrões apresentados no capítulo 5. Por simplicidade, o tipo *MarketData* será referenciado ao longo da seção como *Market*.

A regra de ativação referente ao tipo de situação *Sit1* indica que o mesmo deve ser ativado sempre que o valor do campo *ticker*, encontrado nos eventos do tipo *Market* (indicado na cláusula *from*), corresponder a número par (restrição na cláusula *where*, “*ticker%2 = 0*”). Como os valores do campo *ticker* são atribuídos de forma sequencial, metade dos eventos do tipo *Market* possuem valor do atributo *ticker* representado por um número par. Ou seja, metade dos eventos gerados são responsáveis por ativar uma situação do tipo *Sit1*. A regra de desativação referente a esse tipo é oposta à regra ativação, indicando que o mesmo deve ser desativado ao ser detectado um evento do tipo *Market* cujo atributo *ticker* possui um valor ímpar. Da mesma forma, metade dos eventos gerados são responsáveis por desativar uma situação do tipo *Sit1*. Assim, espera-se que a quantidade de eventos gerados do tipo de situação *Sit1* (somando eventos de ativação e desativação) seja igual à quantidade de eventos do tipo *Market* processados pelo servidor.

	Sit1	Sit2
ATIVAÇÃO	<pre>select marketData from Market as marketData where ticker%2 = 0</pre>	<pre>select sit1 from Sit1 as sit1 where Sit1.activated = true</pre>
DESATIVAÇÃO	<pre>select marketData from Sit1.std:unique(id) as Sit1, Market.std:lastevent() as marketData where Sit1.activated = true and not(marketData.ticker%2 = 0)</pre>	<pre>select sit1 from Sit2.std:unique(id) as Sit2, Sit1.std:lastevent() as Sit1 where Sit2.activated is true and (Sit2.sit.id = Sit1.id and Sit1.activated is not true)</pre>

Figura 73. Regras de ativação e desativação dos tipos de situação *Sit1* e *Sit2*

A regra de ativação referente ao tipo de situação *Sit2* faz-se simples, indicando que ele deve ser ativado quando uma situação do tipo *Sit1* for ativada. De forma análoga, a regra de desativação indica que o tipo de situação *Sit2* deve ser desativado quando a instância do tipo de situação *Sit1* que levou a sua ativação for dada como desativada. Assim, espera-se que a quantidade de eventos gerados

correspondentes ao tipo de situação *Sit2* (somando eventos de ativação e desativação) seja igual à quantidade de eventos gerados correspondentes ao tipo de situação *Sit1*.

Tomando como base um cenário onde X eventos do tipo *Market* são enviados ao servidor, conforme apresentado na Figura 74, as questões discutidas anteriormente referentes ao número de eventos esperados são sumarizadas pela Figura 75 e pela Figura 76, apresentando o número de eventos esperados utilizando somente a plataforma Esper e utilizando a plataforma Esper juntamente com a plataforma SIMPLE, respectivamente.

$$X \text{ eventos do tipo } Market = \frac{X}{2} \text{ eventos com valor de } ticker \text{ par} + \frac{X}{2} \text{ eventos com valor de } ticker \text{ impar}$$

Figura 74. Cenário onde X eventos do tipo *Market* são enviados ao servidor

Conforme a Figura 75, na primeira etapa de testes, utilizando apenas a plataforma Esper, são esperados um total X eventos correspondentes ao número de eventos do tipo *Market* recebidos pelo servidor. Por outro lado, na segunda etapa de testes, ao utilizar a plataforma Esper juntamente com a plataforma SIMPLE são esperados um número de eventos totais igual a três vezes o número de eventos encontrados na primeira etapa de testes, correspondentes ao número de eventos do tipo *Market* recebidos pelo servidor somados ao número de eventos de ativação e desativação gerados pela plataforma SIMPLE referentes aos tipos de situação *Sit1* e *Sit2*.

Número de eventos utilizando somente a plataforma Esper

$$\text{Total de eventos} = \frac{X}{2} \text{ eventos com valor de } ticker \text{ par} + \frac{X}{2} \text{ eventos com valor de } ticker \text{ impar} = X \text{ eventos}$$

Figura 75. Número de eventos esperados utilizando apenas a plataforma Esper

Uma vez que as duas etapas de testes foram executadas, os resultados foram comparados a fim de mensurar o impacto na plataforma Esper gerado pela utilização da plataforma SIMPLE sobre a mesma. A seção seguinte apresenta e discute os resultados obtidos a partir dessa comparação.

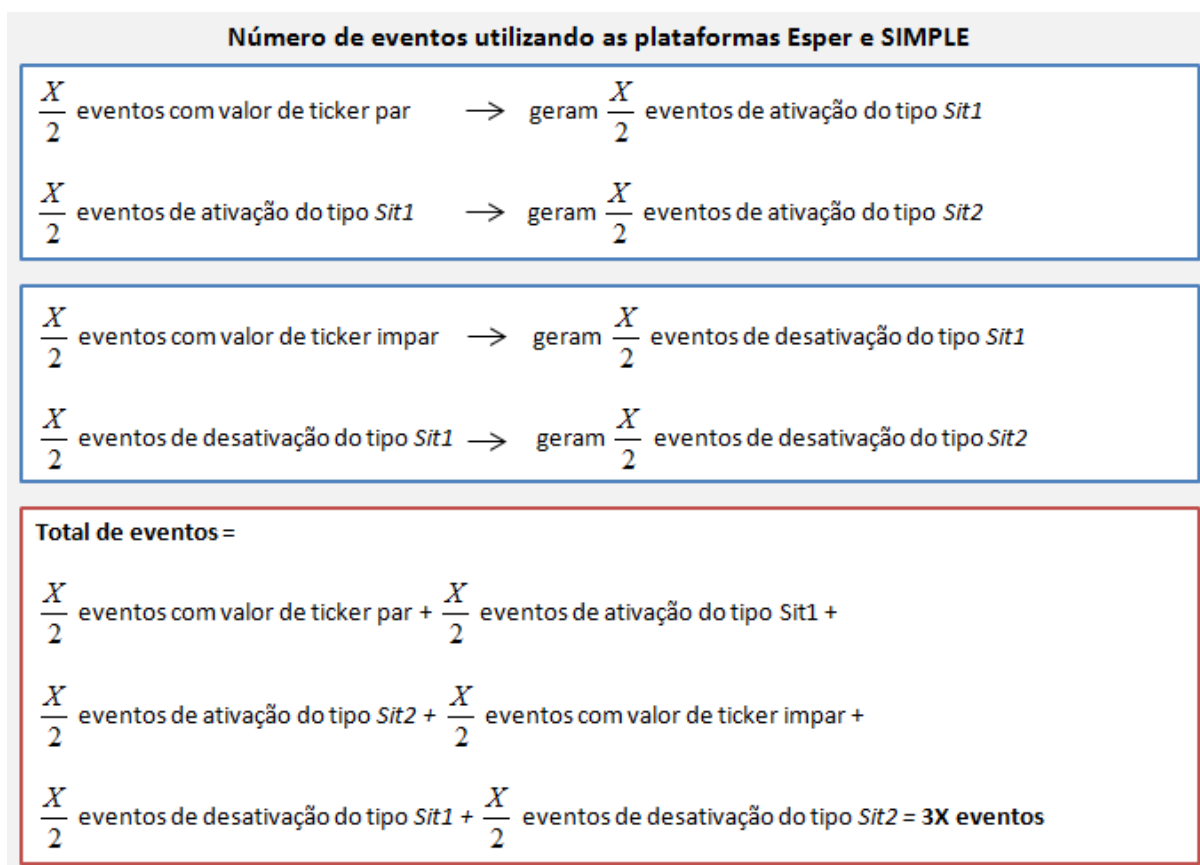


Figura 76. Número de eventos esperados utilizando as plataformas Esper e SIMPLE

7.2 RESULTADOS

As figuras apresentadas nesta seção correspondem aos resultados obtidos após a realização das duas etapas de testes. Em cada figura estão contidos quatro gráficos: dois gráficos ilustram os valores das latências médias obtidas em cada uma das duas etapas de testes, e dois gráficos ilustram os valores médios obtidos para a vazão em cada uma das duas etapas de testes. Os gráficos correspondentes à primeira etapa de testes são exibidos ao lado esquerdo, enquanto que os gráficos correspondentes à segunda etapa de testes são exibidos do lado direito.

Cada gráfico representa uma bateria de testes, contendo 56 testes, e seu eixo horizontal possui como escala o número do teste em questão. Isso corresponde a dizer que cada ponto contido em um gráfico representa o valor médio (de latência ou vazão) obtido após a execução de um teste unitário com duração de 5 segundos. O

eixo vertical apresenta sua escala em nanosegundos (ns) para a latência, e em número de eventos por segundo (evt/s) para a vazão. Os gráficos referentes aos valores médios de vazão correspondentes à segunda etapa de testes apresentam também informações sobre o número de eventos médios gerados pela plataforma Esper, representados pela cor roxa, e o número de eventos médios gerados pela plataforma SIMPLE, representados pela cor azul.

Durante o início da segunda bateria de testes foi detectado um problema na especificação dos tipos de situação *Sit1* e *Sit2*. O uso das janelas “*std:lastevent()*” e “*win:unique(id)*”, utilizadas em ambas as regras de desativação, levaram a um consumo de memória excessivo. Isso ocorre pois, para cada último evento (*std:lastevent()*) e cada *id* diferente (*win:unique(id)*) é aplicado um *join*, realizando um produto cartesiano entre os diferentes fluxos de eventos. Cada instância de um tipo de situação criada recebe um *id* novo, originando uma nova janela do tipo *win:unique(id)*. O envio de 100.000 evt/s corresponde à ativação e criação de 50.000 novas instâncias do tipo de situação *Sit1* por segundo e, conseqüentemente, a criação de 50.000 janelas por segundo. Ainda, cada ativação de uma situação do tipo *Sit1* ocasiona em uma ativação de uma situação *Sit2*, totalizando a criação 100.000 de janelas por segundo. Todas essas janelas são armazenadas em memórias, uma vez que, eventualmente, podem ser reavaliadas posteriormente.

Por volta do teste unitário número 20 da primeira bateria, notou-se que praticamente toda memória disponível para a JVM era utilizada, impactando radicalmente no desempenho da plataforma. Isso tornou inviável qualquer tipo de análise sobre os dados apresentados, uma vez que a vazão rapidamente tendia a zero, apresetando, conseqüentemente, valores exorbitantes para a latência. Por esta razão, optou-se por alterar as definições das regras de desativação dos tipos de situação *Sit1* e *Sit2*. A Figura 77 ilustra as novas definições. Embora possuam sintaxes distintas, para um cenário totalmente restrito e controlado conforme para a realização dos testes, ambas mantêm a mesma semântica e comportamento apresentado na seção 7.1.

As subseções seguintes apresentam os resultados e análises referentes a cada uma das cinco baterias de testes, a partir da comparação dos valores obtidos ao final das duas etapas de testes.

	Sit1	Sit2
D E S A T I V A Ç Ã O	<pre>select marketData from Market as marketData where not(ticker%2 = 0)</pre>	<pre>select sit1 from Sit1 as sit1 where not(Sit1.activated = true)</pre>

Figura 77. Novas definições para as regras de desativação dos tipos de situação *Sit1* e *Sit2*

7.2.1 Primeira bateria

A Figura 78 ilustra os comparativos entre os resultados obtidos após as duas etapas de testes, referente à primeira bateria de testes. Nessa bateria, foram seguidos os passos apresentados na metodologia utilizando apenas um cliente local.

Conforme é possível observar no gráfico correspondente à vazão utilizando apenas a plataforma Esper (*i.e.*, utilizando o *Esper Benchmark Kit*), o valor médio obtido para a vazão consiste em aproximadamente 60.000 evt/s. A utilização dos testes de forma local, e não distribuída, pode ter contribuído para esse resultado, uma vez que servidor e cliente competiam pelos mesmos recursos. A utilização de outras configurações da JVM pode contribuir para uma melhor vazão, uma vez que as configurações utilizadas, conforme discutido anteriormente, foram escolhidas de forma empírica. Ambos os testes distribuídos e análise das configurações da JVM devem ser melhor trabalhadas e estudadas em trabalhos futuros.

Ao comparar os dois gráficos de vazão, os eventos gerados pela plataforma Esper (*i.e.*, gerados pelo cliente) mantiveram o valor obtidos ao utilizar apenas o *Esper Benchmark Kit*, mantendo-se em torno de aproximadamente 60.000 evt/s. Por sua vez, os eventos gerados pela plataforma SIMPLE correspondem ao dobro desse número de eventos, mantendo-se em torno de aproximadamente 120.000 evt/s, ou seja, 60.000 evt/s para cada tipo de situação. Dessa forma, utilizando a plataforma SIMPLE a vazão média obtida manteve-se em aproximadamente 180.000 evt/s. Conforme discutido na seção 7.1.2, era esperado que o número de eventos

utilizando a plataforma Esper juntamente com a plataforma SIMPLE correspondesse a três vezes o número de eventos obtidos ao utilizar somente a plataforma Esper. Esse fato foi confirmado durante a realização da primeira bateria de testes.

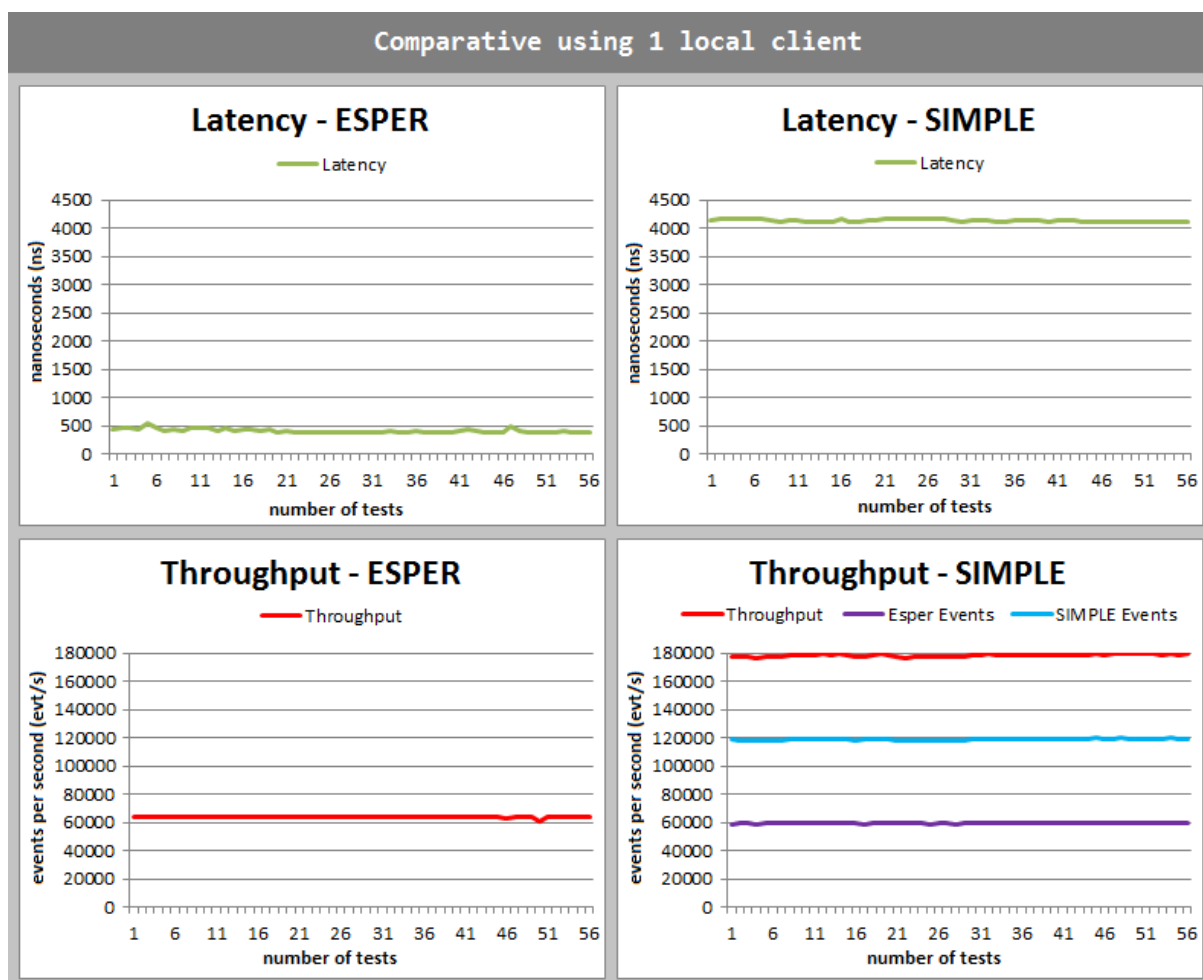


Figura 78. Comparativo entre os resultados da primeira bateria de testes

A latência, por sua vez, utilizando a plataforma SIMPLE, atingiu valores oito vezes maiores, mantendo-se em aproximadamente 4000 ns, ao passo que usando apenas a plataforma Esper os valores mantiveram-se em aproximadamente 500 ns. Um valor maior para a latência média é esperado, uma vez que, além do aumento de processamento introduzido pelo uso da plataforma SIMPLE para detectar e gerenciar o ciclo de vida das situações, ocorreu um aumento significativo no número de eventos processado pela plataforma Esper, sendo esse número três vezes maior.

7.2.2 Segunda bateria

A Figura 79 ilustra os comparativos entre os resultados obtidos após as duas etapas de testes, referente à segunda bateria de testes. Nessa bateria, foram seguidos os passos apresentados na metodologia utilizando dois clientes locais.

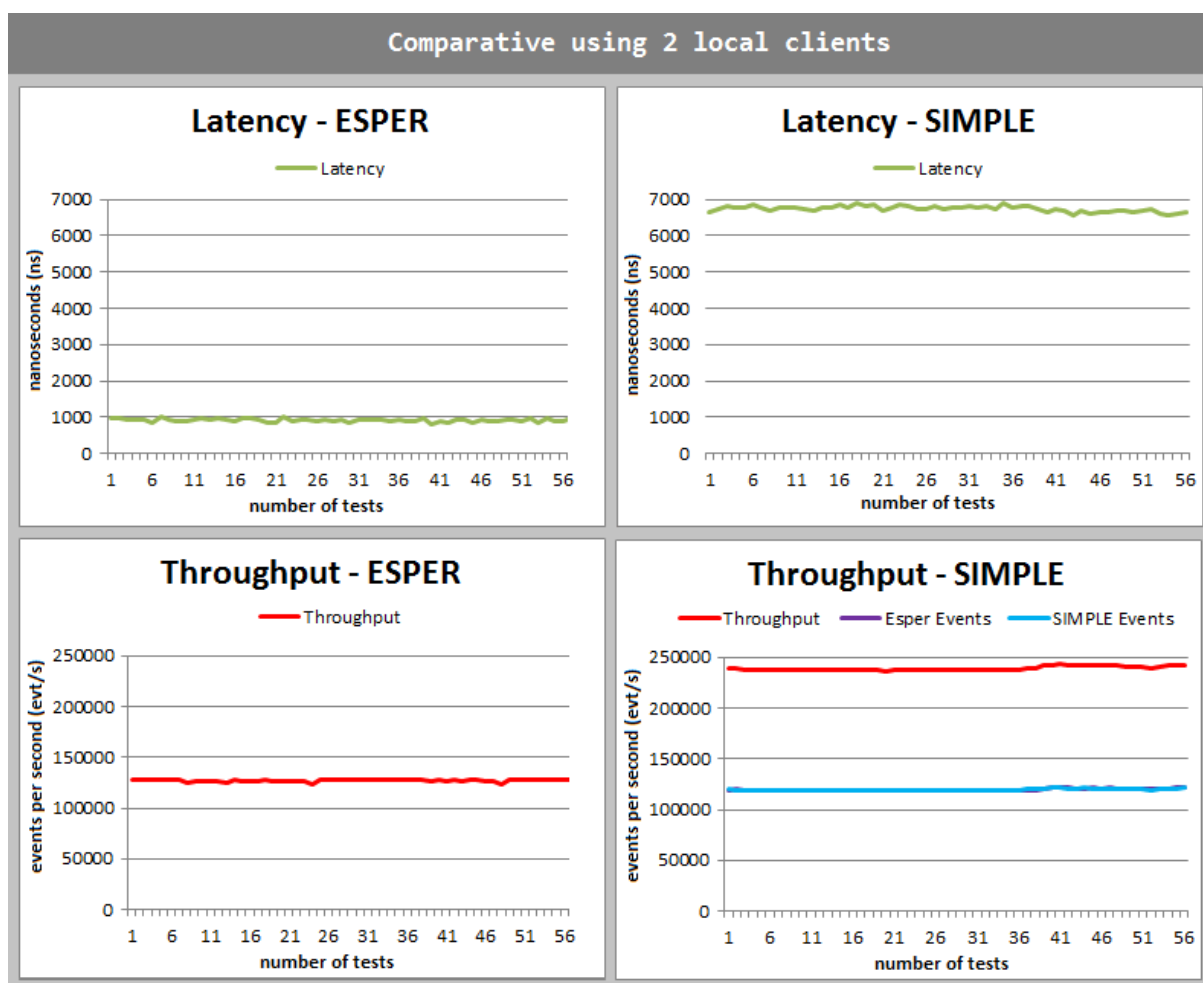


Figura 79. Comparativo entre os resultados da segunda bateria de testes

Ao utilizar dois clientes, pode-se perceber que o valor da vazão média, utilizando apenas a plataforma Esper, manteve-se aproximadamente em 130.000, aproximadamente o dobro dos valores obtidos durante a primeira bateria de testes. Por sua vez, a vazão média utilizando a plataforma SIMPLE mostrou-se diferente da primeira bateria de testes. Em vez do número de eventos criados pela plataforma SIMPLE corresponder ao dobro de eventos criados pela plataforma Esper, ambos os valores foram exatamente iguais. Dessa forma, ambas as linhas foram sobrepostas,

de forma que a linha roxa, representando os eventos da plataforma Esper, foi ocultada pela linha azul, representando os eventos da plataforma SIMPLE.

Acreditamos que o número de eventos não corresponde ao dobro pelo fato dos clientes terem sido executados em paralelo, uma vez que cada um é representado por uma *thread* diferente, é possível que ambas tenham sido executadas paralelamente, tendo em vista a utilização de um computador de quatro núcleos. Considerando essa possibilidade, eventos de ambos os clientes possivelmente foram gerados e enviados à plataforma Esper praticamente ao mesmo instante. Como o servidor recebeu dois eventos com o mesmo valor para o atributo *ticker* ao mesmo tempo (ou com uma diferença de tempo ínfima), apenas um dos eventos levou a ativação cada situação, sendo o outro ignorado e descartado pela plataforma SIMPLE por representar um evento de ativação referente a uma situação que já encontra-se ativa. O mesmo raciocínio é válido para os eventos com valor ímpar para o atributo *ticker*, que levariam à desativação da situação.

Com a adição de um novo cliente, a latência média assumiu um comportamento linear. Em ambos os casos, seu valor encontrou-se dobrado em comparação aos valores obtidos durante a primeira bateria de testes, mantendo-se em aproximadamente 7000 ns com a utilização da plataforma SIMPLE e em 1000 ns com a utilização apenas da plataforma Esper. Apesar disso, proporcionalmente ao respectivo valor de vazão, a latência utilizando a plataforma SIMPLE apresentou um aumento superior, uma vez que sua vazão total teve aumento apenas de 30% em relação aos valores obtidos pela primeira bateria.

7.2.3 Terceira bateria

A Figura 80 ilustra os comparativos entre os resultados obtidos após as duas etapas de testes, referentes à segunda bateria de testes. Nessa bateria, foram seguidos os passos apresentados na metodologia utilizando três clientes locais.

Durante essa bateria de testes, foi possível observar a ocorrência de alguns picos negativos durante a análise da vazão utilizando apenas a plataforma Esper. Acreditamos que isso possa ter sido causado por algum mecanismo interno, por

exemplo, *garbage collector* na JVM ou alguma otimização interna ativada pela plataforma Esper e que teria consumido momentaneamente alguns recursos. Essas suspeitas são reforçadas pelo fato de que tais picos ocorreram em intervalos regulares, e mantiveram-se mesmo com a repetição dos testes. Porém, era esperado que o mesmo comportamento fosse refletido ao utilizar essa bateria de testes juntamente com a plataforma SIMPLE, fato este que não ocorreu. O número de eventos gerados pela plataforma SIMPLE manteve-se aproximadamente o dobro do número de eventos gerados pela plataforma Esper, conforme o esperado.

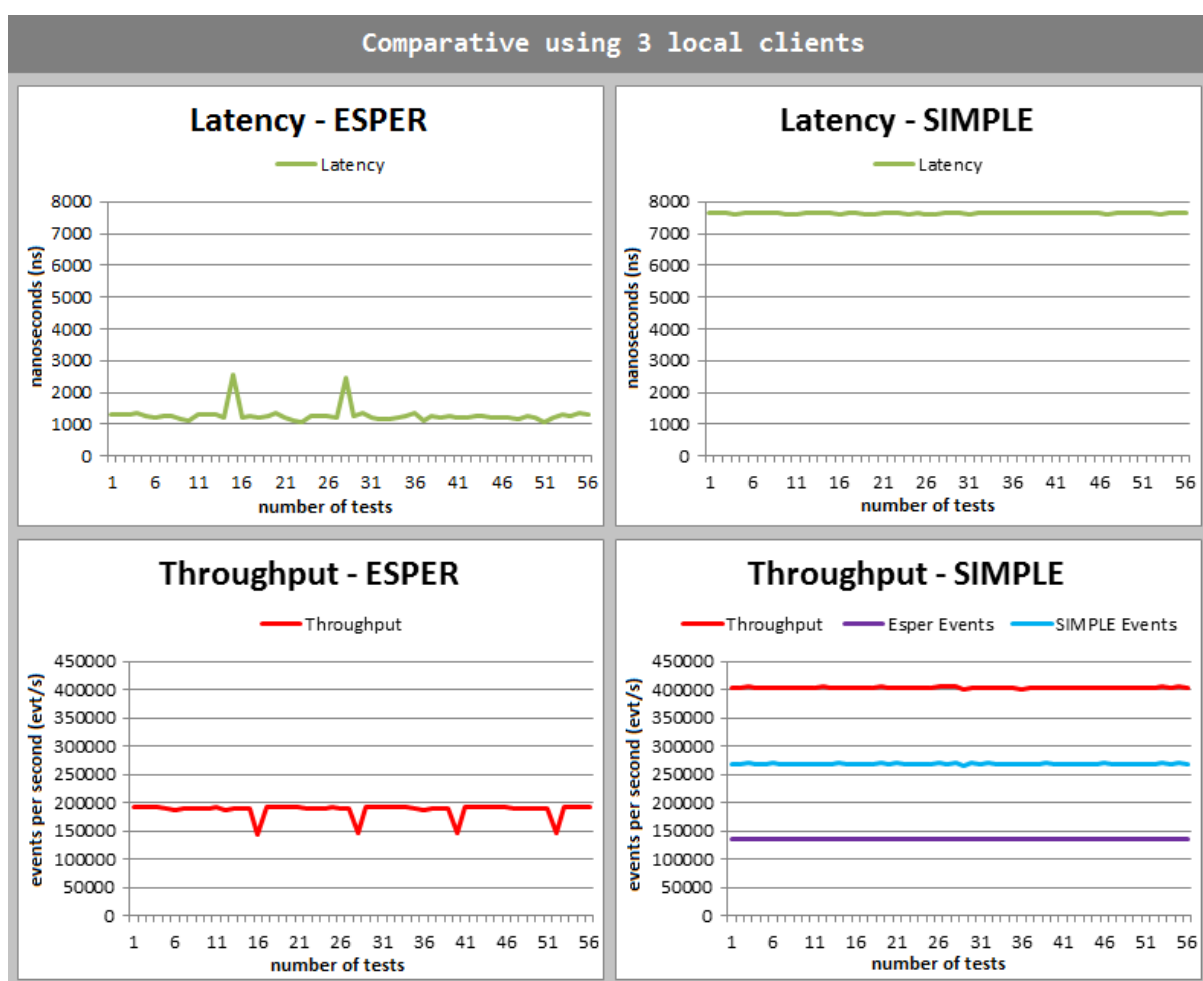


Figura 80. Comparativo entre os resultados da terceira bateria de testes

Partindo do pressuposto que tais picos foram motivados pelo *garbage collector*, a sua não ocorrência utilizando a plataforma SIMPLE faz-se coerente. Utilizando a plataforma SIMPLE, os eventos recebidos pela plataforma Esper continuam a ser utilizados pelos eventos correspondentes às ativações e desativações dos tipos de situações, de forma que o *garbage collector* não é acionado. Isso não ocorre

utilizando somente a plataforma Esper, onde os eventos recebidos são processados uma vez e não são utilizados novamente até o fim da execução, de forma que o *garbage collector* deve ser acionado para liberar os objetos correspondentes (não mais utilizados) da memória.

Chama atenção os valores obtidos para as latências médias durante essa bateria de testes. Considerando a evolução dos valores obtidos pelas duas primeiras baterias, esperava-se que a introdução do terceiro cliente lavaria a um acréscimo na latência média, conforme ocorrido anteriormente. Porém, ambas mantiveram seus valores muito próximos aos valores obtidos na segunda bateria, tanto com a utilização da plataforma SIMPLE quanto utilizando apenas a plataforma Esper.

7.2.4 Quarta bateria

A Figura 81 ilustra os comparativos entre os resultados obtidos após as duas etapas de testes, referente à segunda bateria de testes. Nessa bateria, foram seguidos os passos apresentados na metodologia utilizando quatro clientes locais.

A partir da adição do quarto cliente, pode-se observar uma queda no desempenho da plataforma. Na segunda etapa os valores da vazão, mantiveram-se praticamente os mesmos, se comparados com a segunda etapa da bateria anterior. As latências, por sua vez, obtiveram algum aumento, indicando que a plataforma demorou mais tempo para processar cada evento, levando em conta praticamente o mesmo número de eventos gerados por segundo.

Como nessa bateria existem 4 clientes e um servidor sendo executados em um mesmo computador, torna-se impossível executar todos os componentes em paralelo utilizando um computador que disponha de apenas quatro núcleos de processamento. Dessa forma, para um número de clientes maior do que 3, de forma local, é esperado que o desempenho seja prejudicado, uma vez que é impossível obter um paralelismo real. Possivelmente, nas outras baterias não existia paralelismo real durante toda a execução, uma vez que existem outros processos do próprio sistema operacional sendo executados concorrentemente e competindo por recursos. Porém, em alguns instantes do tempo, tal paralelismo fazia-se possível.

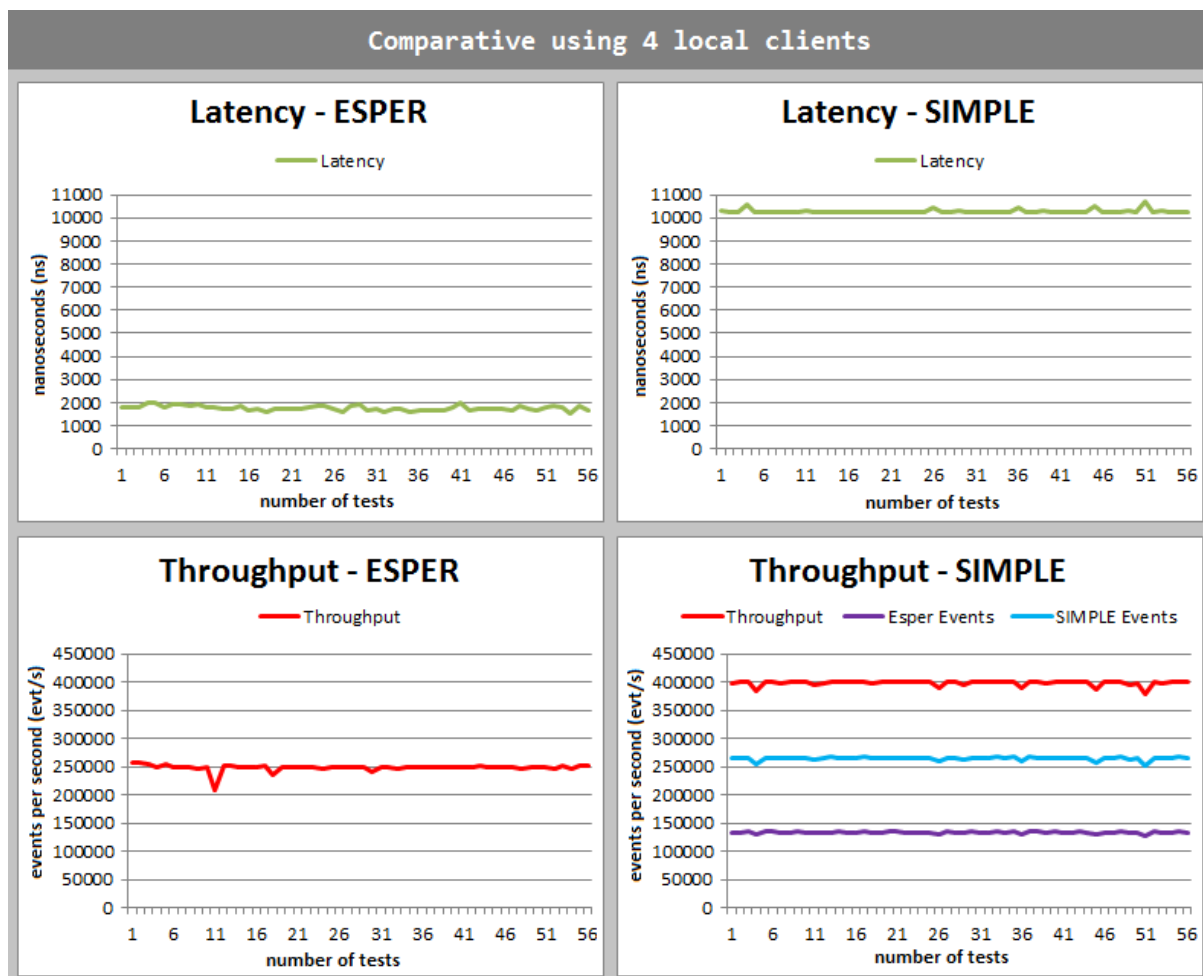


Figura 81. Comparativo entre os resultados da quarta bateria de testes

Os testes realizados com quatro ou mais clientes foram prejudicados pela configuração local. Para um melhor entendimento desse comportamento, seria necessário a realização dos testes de forma distribuída, com recursos suficientes para cada um dos clientes e do servidor. De forma local, tais testes foram prejudicados pelas limitações de nosso ambiente físico de testes. A realização de tais testes é indicada para trabalhos futuros.

7.2.5 Quinta bateria

A Figura 82 ilustra os comparativos entre os resultados obtidos após as duas etapas de testes, referente à segunda bateria de testes. Nessa bateria, foram seguidos os passos apresentados na metodologia utilizando dez clientes locais.

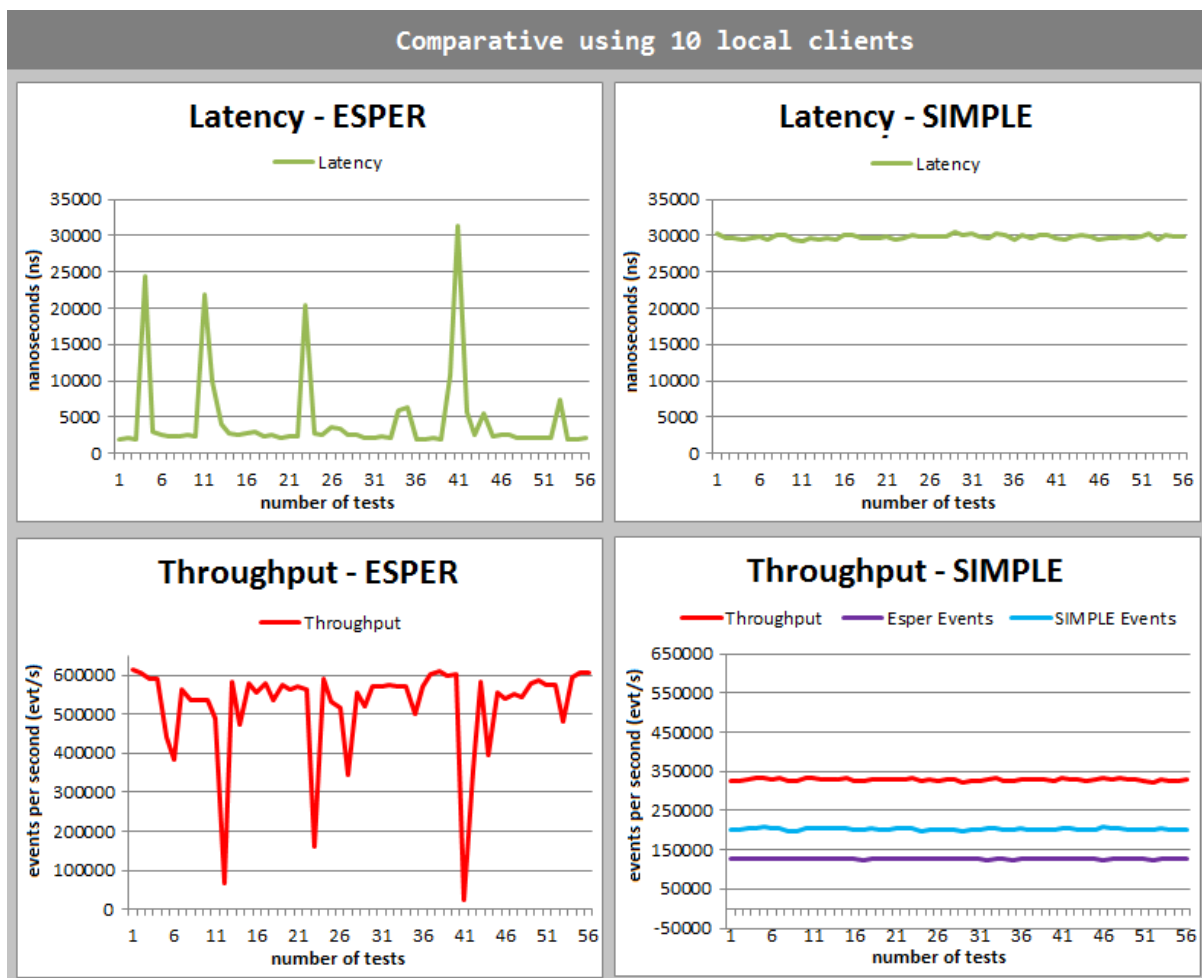


Figura 82. Comparativo entre os resultados da quinta bateria de testes

A última bateria foi realizada apenas para comprovar que, a partir da utilização de quatro clientes, foi encontrada uma limitação de *hardware*, fato este previamente esperado. Utilizando apenas a plataforma Esper, foi possível alcançar valores superiores de vazão, atingindo valores de aproximadamente 600.000 evt/s. Porém esses valores mostram-se instáveis, com diversos picos negativos, chegando em alguns momentos a alcançar uma vazão menor que 100.000 evt/s. Esse comportamento refletiu-se na latência média, que em momentos apresentava valores próximos a 4.000 ns, e logo em seguida valores que ultrapassavam 30.000 ns.

Já utilizando a plataforma SIMPLE, os valores mantiveram-se estáveis, porém com desempenho abaixo do esperado. Os valores da vazão média foram semelhantes, mas ligeiramente piores que os apresentados pela quarta bateria. Por

sua vez, a latência média de aproximadamente 30.000 ns expõe o quão pior foi o desempenho obtido.

7.3 CONSIDERAÇÕES DO CAPÍTULO

Ao longo deste capítulo foram apresentados os testes de desempenho realizados com o intuito de mensurar o impacto da plataforma SIMPLE sobre a plataforma Esper. Foi apresentada a metodologia proposta para a elaboração de tais testes. Além disso, os resultados foram ilustrados, comparados e discutidos.

Os resultados obtidos para vazão mantiveram-se dentro do esperado com a utilização de até três clientes de forma local, onde o número de eventos gerados pela plataforma SIMPLE correspondeu ao dobro gerado pela plataforma Esper, totalizando um número de eventos três vezes maior quando comparado com a realização dos testes utilizando apenas a plataforma Esper. Apenas na segunda bateria isso não ocorreu, porém a execução dos clientes em paralelo por parte do sistema operacional pode explicar tal comportamento.

Ao utilizar quatro clientes ou mais, as configurações de *hardware* utilizadas mostraram-se, aparentemente, insuficientes para usufruir de toda a capacidade de processamento da plataforma Esper, sendo necessário um estudo futuro fazendo uso de um ambiente totalmente distribuído. Os testes finais apresentaram uma possível limitação da plataforma SIMPLE. Sem a plataforma SIMPLE foi possível atingir valores de vazão, mesmo que momentâneos, de aproximadamente 600.000 evt/s na quinta bateria de testes. Com a utilização da plataforma SIMPLE, porém, a maior vazão alcançada foi de aproximadamente 400.000 ao longo de todas as baterias de testes. Uma vez que a plataforma Esper, utilizando as mesmas configurações de *hardware* e *software* atingiu 600.000 evt/s, era esperado que a plataforma SIMPLE atingisse resultados semelhantes, mesmo que com altas variações ou altos valores de latência associados, fato que não ocorreu. Uma possível explicação consiste no fato do *SituationManagerListener* ser uma estrutura bloqueante. Mesmo que duas situações sejam ativadas ao mesmo tempo, essas ativações não ocorrem em paralelo, sendo cada chamada ao

SituationManagerListener realizada de forma sequencial. Para um número alto de eventos, e consequentemente, de situações ativadas (e desativadas) por segundo, é necessário que tal componente trabalhe em paralelo para lidar de forma melhor com um alto volume de dados.

Quanto aos valores de latência adquiridos, pôde-se medir o impacto da plataforma SIMPLE sobre a plataforma Esper, onde, considerando o ambiente local, manteve-se cerca de oito vezes maior enquanto os testes puderam ser realizados sem sofrer grandes limitações de *hardware* (até três clientes). Porém, não é possível afirmar se os valores apresentados são satisfatórios ou não, uma vez que não foi encontrado na literatura plataformas semelhantes que apoiem a utilização de situações em sistemas de CEP conforme proposta na metodologia apresentada, de forma que os resultados obtidos pudessem ser comparados com demais plataformas.

O capítulo seguinte apresenta as considerações finais do presente trabalho, bem como discute possíveis trabalhos futuros.

8 CONSIDERAÇÕES FINAIS

8.1 CONCLUSÕES

Este trabalho propôs uma metodologia para auxiliar a especificação, detecção e processamento de situações em sistemas de CEP, tanto em tempo de projeto quanto em tempo de execução. Esse objetivo geral foi subdividido em quatro objetivos específicos: (i) definir conceitos que sirvam como fundamentação para a realização de situações no domínio de CEP; (ii) desenvolver uma plataforma CEP que permita a especificação de situações em tempo de projeto e em tempo de execução; (iii) transformar os modelos contendo as definições das situações do domínio, descritos durante a etapa de *design*, em código específico da plataforma de situações desenvolvida no segundo objetivo e (iv) analisar o desempenho da plataforma de situações desenvolvida no segundo objetivo.

O primeiro objetivo resultou em um modelo de detecção de situações em CEP (seção 4.1), que proporcionou um melhor entendimento a respeito da utilização de situações em sistemas de CEP. Uma vez definidos os conceitos necessários para o entendimento das áreas de sistemas CEP e de aplicações sensíveis a contexto, foi desenvolvida a plataforma SIMPLE (*Situation Mapping Layer for Esper*) (capítulo 4), a fim de prover suporte em tempo de execução para realização de situações (*i.e.*, a detecção e processamento de situações) de acordo com o segundo objetivo. Essa plataforma foi desenvolvida utilizando como base a plataforma Esper.

A plataforma SIMPLE gerencia o ciclo de vida das situações, monitorando continuamente os fluxos de eventos a fim de verificar se as condições que levaram à ativação (ou desativação) de uma situação continuam satisfeitas., além de impedir a execução de regras de ativação referentes às situações já ativas e a execução de regras de desativação referentes às situações já desativadas. A plataforma também permite composicionalidade e integridade das situações. Sua utilização possibilita retirar do usuário toda a complexidade envolvida no processamento e detecção das

situações, permitindo que o usuário concentre-se em questões específicas do seu domínio.

Uma vez implementada tal plataforma, foram executados testes de desempenho com o intuito de mensurar o impacto da plataforma SIMPLE sobre a plataforma Esper. Foram elaboradas cinco baterias de testes, cada uma utilizando um número diferente de clientes. Inicialmente, as baterias foram executadas utilizando apenas a plataforma Esper. Posteriormente, as baterias foram repetidas, com acréscimo de dois tipos de situações, utilizando a plataforma SIMPLE. Após ambas as execuções, os resultados foram comparados e analisados. Inicialmente os resultados ficaram um pouco abaixo do esperado, apresentando resultados de vazão de 60.000 evt/s, enquanto o esperado era uma taxa de 100.000 evt/s. Acreditamos que a realização de testes locais (utilizando apenas um computador) e as configurações da JVM utilizadas podem ter influenciado tal resultado. Esses pontos devem ser melhor explorados em trabalhos futuros, principalmente com a realização de testes distribuídos. Além disso, é desejada a elaboração de testes utilizando tipos de situações mais complexos, a fim de avaliar a influência de operadores como *followed by* e *every*.

Durante a realização dos testes, foram detectados alguns problemas: (i) a utilização de uma estrutura *hash* para armazenar as instâncias dos tipos de situações geradas, levando a um aumento considerável na utilização da memória; (ii) a utilização de apenas um componente do tipo *SituationManager*, que consiste em um ponto único de falhas, além de uma possível sobrecarga devido à enorme quantidade de requisições e (iii) a utilização das janelas *std::unique(id)* e *std::lastevent()*, causando um consumo excessivo de memória, de forma que foi necessária a alteração dos tipos das situações definidas para os testes. Todos esses pontos devem ser melhor estudados, principalmente a utilização das janelas, uma vez que os padrões definidos neste trabalho para a representação de tipos de situação fazem uso delas.

Com o intuito de prover suporte em tempo de projeto foi utilizada a linguagem gráfica SML (seção 2.4), permitindo que desenvolvedores modelem situações num alto nível de abstração. Durante o desenvolvimento da metodologia proposta, verificou-se que as regras EPL especificadas para definir situações (simples e

compostas) eram muito complexas e, por vezes, inviáveis de serem definidas manualmente. A fim de resolver este problema, a metodologia utilizou técnicas de desenvolvimento orientado a modelos, para permitir (i) a especificação de situações em um alto nível de abstração, e (ii) o mapeamento automático destas especificações para código em uma EPL.

A partir dos modelos de situações especificados em SML, foram definidas transformações dos modelos para código a ser executado diretamente pela plataforma SIMPLE, permitindo que a plataforma detecte e gerencie de forma adequada as situações modeladas, com base no terceiro objetivo. As regras de transformação foram definidas a partir de quatro padrões responsáveis por cobrir as principais definições de tipos de situações. Dentre esses padrões, dois apresentaram-se complexos: o padrão “Situação Composta” (seção 5.4) e o padrão “Exists” (seção 5.3). O padrão “Situação Composta” apresentou complexidade fatorial em sua definição, tornando inviável sua especificação de forma manual, motivando a utilização de uma abordagem de desenvolvimento orientado a modelos. Já o padrão “Exists”, mostrou-se um caso particular, de forma que durante o desenvolvimento deste trabalho não foi possível generalizá-lo a fim de permitir sua combinação com o padrão “Situação Composta”. Esse aspecto consiste numa limitação do presente trabalho, de forma que para contorná-la, faz-se necessário definir uma situação intermediária, conforme discutido na seção 5.3.

Para verificar a viabilidade da abordagem proposta, considerando tanto os aspectos de tempo de projeto quanto de tempo de execução, foi utilizado um cenário voltado para o ambiente de fraudes bancárias (seção 6.1). A partir dos modelos SML descrevendo as situações do cenário proposto foi gerado automaticamente código para processar, detectar e gerenciar o ciclo de vida das situações utilizando a plataforma SIMPLE. A fim de validar o código gerado e verificar a viabilidade da utilização de situações em aplicações reais, foi implementada uma aplicação para simular um ambiente bancário. Apesar disso, o presente trabalho reconhece a necessidade em utilizar outros cenários, mais complexos, para verificar se os padrões propostos neste trabalho são suficientes para cobrir todos os tipos de situações possíveis e, ainda, se as regras de transformação cobrem todas as possibilidades de combinação entre esses padrões.

8.2 TRABALHOS FUTUROS

Como trabalhos futuros, ficam as seguintes sugestões:

- Permitir a criação de mais de uma instância do componente *SituationManager*, uma vez que o mesmo torna-se um ponto único de falhas e pode não ser capaz de lidar com uma enorme quantidade de dados por segundo. A utilização de várias instâncias permite processamento em paralelo, porém é necessário manter todas as instâncias sincronizadas de forma a garantir a integridade das situações;
- Estudar o uso da estrutura *hash* para armazenar as instâncias dos tipos de situação, de forma a diminuir a quantidade de memória utilizada. Deve-se considerar distribuir tal estrutura (com atenção aos aspectos relacionados à sincronia), alterar o tipo de estrutura ou armazenar instâncias antigas em fontes de dados externas, como bancos de dados;
- Melhorar a regra de transformação para o padrão *Exists*, uma vez que a mesma restringe o uso de um tipo de situação participante por tipo de situação especificado;
- Avaliar os padrões propostos para transformação de modelos de situação SML em código a ser executado na plataforma SIMPLE em outros cenários, a fim de verificar a necessidade de definir novas regras ou padrões;
- Estudar o uso de operadores temporais de Allen em sistemas orientados a eventos: (i) avaliar quais operadores podem ser aplicados ao utilizar eventos para a detecção de situações, (ii) avaliar as alterações necessárias nos padrões propostos para permitir a utilização dos operadores de Allen e, (iii) uma vez definidas as alterações necessárias nos padrões propostos, implementar as transformações equivalentes;
- Repetir os testes de desempenho, porém de forma distribuída e utilizando melhores configurações de *hardware*;

- Estudar o uso das configurações da JVM a fim de garantir o melhor desempenho da plataforma Esper em cada situação, uma vez que as configurações utilizadas neste trabalho foram selecionadas a partir de resultados empíricos;
- Realizar testes de desempenho mais complexos, variando não só o número de clientes como o número de eventos, de *queries*, além de utilizar tipos de situações que envolvam construções mais complexas, como os operadores *every* e *followed by*. Também é desejado analisar de forma sistemática o uso de memória e de CPU para um melhor entendimento dos resultados obtidos;
- Aplicar a metodologia proposta neste trabalho em diferentes plataformas de CEP;

REFERÊNCIAS

- ABOWD, Gregory D. et al. Towards a better understanding of context and context-awareness. In: **Handheld and ubiquitous computing**. Springer Berlin Heidelberg, 1999. p. 304-307.
- ALLEN, James F. Maintaining knowledge about temporal intervals. **Communications of the ACM**, v. 26, n. 11, p. 832-843, 1983.
- ALMEIDA, João Paulo A. Model-driven design of distributed applications. In: **On the Move to Meaningful Internet Systems 2004: OTM 2004 Workshops**. Springer Berlin Heidelberg, 2004. p. 854-865.
- ALMEIDA, João Paulo; PIRES, Luís Ferreira; VAN SINDEREN, Marten. Costs and benefits of multiple levels of models in MDA development. In: **Proceedings of the Second European Workshop on Model Driven Architecture (MDA) with an emphasis on Methodologies and Transformations “Computer Science at Kent**. 2004. p. 12-21.
- BAPTISTA, G. L. B. et al. On-line Detection of Collective Mobility Patterns through Distributed Complex Event Processing. **Monografias em Ciência da Computação-MCC**, v. 12, p. 2013, 2013.
- BARWISE, Jon. **The situation in logic**. Center for the Study of Language (CSLI), 1989.
- BRUNS, Ralf et al. DS-EPL: Domain-specific event processing language. In: **Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems**. ACM, 2014. p. 83-94.
- COSTA, Patrícia. **Architectural support for context-aware applications: from context models to services platforms**. University of Twente, 2007.
- COSTA, Patricia Dockhorn et al. A model-driven approach to situations: Situation modeling and rule-based situation detection. In: **Enterprise Distributed**

Object Computing Conference (EDOC), 2012 IEEE 16th International.
IEEE, 2012. p. 154-163.

COSTA, Patricia Dockhorn et al. Situations in Conceptual Modeling of Context.
In: **EDOC Workshops**. 2006. p. 6.

CUGOLA, Gianpaolo; MARGARA, Alessandro. Processing flows of information: From data stream to complex event processing. **ACM Computing Surveys (CSUR)**, v. 44, n. 3, p. 15, 2012.

DEY, Anind K. Understanding and using context. **Personal and ubiquitous computing**, v. 5, n. 1, p. 4-7, 2001.

DOMÍNGUEZ, Eladio et al. A survey of UML models to XML schemas transformations. In: **Web Information Systems Engineering–WISE 2007**. Springer Berlin Heidelberg, 2007. p. 184-195.

ECKERT, Michael; BRY, François. Complex event processing (CEP). **Informatik-Spektrum**, v. 32, n. 2, p. 163-167, 2009.

ETZION, Opher; NIBLETT, Peter. **Event processing in action**. Manning Publications Co., 2010.

EUGSTER, Patrick Th et al. The many faces of publish/subscribe. **ACM Computing Surveys (CSUR)**, v. 35, n. 2, p. 114-131, 2003.

FELTRINELLI, Francesco. Human oriented event processing: human interactions in complex event processing. 2011.

FIDLER, Eli et al. The PADRES Distributed Publish/Subscribe System. In: **FIW**. 2005. p. 12-30.

FÜLÖP, Lajos Jenő et al. Survey on complex event processing and predictive analytics. In: **Proceedings of the Fifth Balkan Conference in Informatics**. 2010. p. 26-31.

GRUBER, Tom. What is an Ontology. **WWW Site** <http://www-ksl.stanford.edu/kst/whatis-an-ontology.html> (accessed on 10-10-2015), 1993.

- GUARINO, Nicola. **Formal ontology in information systems: Proceedings of the first international conference (FOIS'98), June 6-8, Trento, Italy.** IOS press, 1998.
- HANSMANN, Uwe et al. **Pervasive computing: The mobile world.** Springer Science & Business Media, 2003.
- HASAN, Souleiman et al. Toward Situation Awareness for the Semantic Sensor Web: Complex Event Processing with Dynamic Linked Data Enrichment. **SSN**, v. 839, p. 69-81, 2011.
- IOT-I. *Final Analysis of Existing IoT Strategic Research Directions and Priorities.* **WWW Site** <http://www.iot-i.eu/public/public-deliverables/d1.6-final-analysis-of-existing-iot-strategic-research-directions-and-priorities/download> (accessed on 01-15-2015), 2012.
- KOKAR, Mięczysław M.; MATHEUS, Christopher J.; BACLAWSKI, Kenneth. Ontology-based situation awareness. **Information fusion**, v. 10, n. 1, p. 83-98, 2009.
- LUCKHAM, David. **The power of events: An introduction to complex event processing in distributed enterprise systems.** Springer Berlin Heidelberg, 2008.
- LUCKHAM, D.; SCHULTE, R. EPTS Event Processing Glossary v2. 0. **Event Processing Technical Society**, 2011.
- MARGARA, Alessandro; CUGOLA, Gianpaolo; TAMBURRELLI, Giordano. Learning from the past: automated rule generation for complex event processing. In: **Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems.** ACM, 2014. p. 47-58.
- MCHEICK, Hamid. Modeling Context Aware Features for Pervasive Computing. **Procedia Computer Science**, v. 37, p. 135-142, 2014.
- MIELKE, Izon Thomaz. *Uma Abordagem Baseada em Modelos para Especificação e Detecção de Situações em Sistemas Sensíveis ao Contexto*, 2013.

OMG. *MDA Guide Version 1.0.1*. WWW Site <http://www.omg.org/cgi-bin/doc?omg/03-06-01> (accessed on 01-21-2015), 2003.

OMG. *UML® Resource Page*. WWW Site <http://www.uml.org/> (accessed on 01-21-2015), 2012.

PEREIRA, Isaac SA; COSTA, Patrícia Dockhorn; ALMEIDA, João Paulo A. A rule-based platform for situation management. In: **Cognitive Methods in Situation Awareness and Decision Support (CogSIMA), 2013 IEEE International Multi-Disciplinary Conference on**. IEEE, 2013. p. 83-90.

RENNERS, Leonard; BRUNS, Ralf; DUNKEL, Jürgen. Situation-aware energy control by combining simple sensors and complex event processing. In: **Workshop on AI Problems and Approaches for Intelligent Environments**. 2012. p. 33.

RICHARDSON, Chris. **POJOs in action**. Dreamtech Press, 2006.

RIZZI RAYMUNDO, Caroline et al. An infrastructure for distributed rule-based situation management. In: **Cognitive Methods in Situation Awareness and Decision Support (CogSIMA), 2014 IEEE International Inter-Disciplinary Conference on**. IEEE, 2014. p. 202-208.

SELIC, Bran. The pragmatics of model-driven development. **IEEE software**, v. 20, n. 5, p. 19, 2003.

SOBRAL, Vinicius M.; ALMEIDA, Joao Paulo A.; DOCKHORN COSTA, Patrícia. Assessing situation models with a lightweight formal method. In: **Cognitive Methods in Situation Awareness and Decision Support (CogSIMA), 2015 IEEE International Inter-Disciplinary Conference on**. IEEE, 2015. p. 42-48.

SWITHINBANK, Peter et al. **Patterns: Model-Driven Development Using IBM Rational Software Architect**. IBM, International Technical Support Organization, 2005.

TIBCO. *Event-Driven SOA: A Better Way to SOA*. **WWW Site**
http://www.tibco.jp/multimedia/wp-event-driven-soa_tcm52-803.pdf
(accessed on 01-15-2015).

ANEXO A – COMPONENTES INTERNOS DA PLATAFORMA SIMPLE

Esse anexo apresenta explicações detalhadas e códigos internos dos componentes do pacote *Situation Utility* da plataforma SIMPLE, discutidos na seção 4.5. Por motivos de simplicidade, são omitidos os métodos *get* e *set* referentes às classes Java apresentadas.

Situation

O componente *Situation* consiste numa classe abstrata responsável por definir os atributos e métodos comuns aos tipos de situações. Toda classe que representa um tipo de situação (*i.e.*, classes do pacote *Situation Class Definitions*) devem estender esta classe. A Figura 83 ilustra, de forma simplificada, a implementação do componente *Situation*. O atributo *activated* (linha 2, Figura 83) é responsável por indicar o *status* das instâncias do tipo de situação correspondente: ativada ou desativada. Uma instância de um tipo de situação é dita ativada caso seu atributo *activated* possua valor igual à *true*. Da mesma forma, uma instância de um tipo de situação é dita desativada caso seu atributo *activated* possua valor igual à *false*. Os atributos *start_time* e *end_time* (linhas 3 e 4, Figura 83) indicam, respectivamente, os tempos de ativação e desativação de cada instância do tipo de situação correspondente. Instâncias ativadas possuem o valor do atributo *end_time* igual à *null*. O atributo *sitName* (linha 5, Figura 83) contém a o nome do tipo de situação correspondente, associando a classe Java ao seu respectivo tipo de situação. Por sua vez, os atributos *epIA* e *epID* (linha 5, Figura 83) são responsáveis por armazenar, respectivamente, as regras de ativação e desativação de cada tipo de situação. Esses três atributos (*sitName*, *epIA* e *epID*) devem ser inicializados no método construtor de cada classe que representa um tipo de situação.

O atributo *id* (linhas 6, Figura 83) é utilizados para controle interno da plataforma SIMPLE, identificando unicamente cada instância de um tipo de situação, de forma a garantir a integridade das situações. Sua atribuição deve ser realizada exclusivamente pela plataforma SIMPLE.

```

1 public abstract class Situation {
2     private boolean activated;
3     private Date start_time;
4     private Date end_time;
5     private String sitName, eplA, eplD;
6     private int id;
7     private String key;
8
9     public abstract Object doActionAtCreateDeactivationEvent();
10    public abstract Situation createNewSit(EventBean event);
11    public Object doActionAtCreateActivationEvent(EventBean[] newEvents){ return null; }
12
13    public void doActionAtDeactivation(EventBean event){
14        System.out.println(
15            "Situation " + getSitName()
16            + " deactivated. ID = " + getId()
17            + " Start Time = " + getStart_time()
18            + "End Time = " + getEnd_time()
19        );
20    }
21
22    public void doActionAtActivation(EventBean event){
23        System.out.println(
24            "Situation " + getSitName()
25            + " activated. ID = " + getId()
26            + " Start Time = " + getStart_time()
27            + "End Time = " + getEnd_time()
28        );
29    }
30 }

```

Figura 83. Componente *Situation*

Imagine, por exemplo, o tipo de situação “febre de João”. Quando uma instância desse tipo de situação é ativada, é enviado à plataforma Esper um evento representando a ativação de uma instância desse tipo situação. Esse evento recebe um número sequencial único para seu atributo *id*, por exemplo, o valor 1. Quando essa mesma instância é desativada, de forma análoga, é enviado à plataforma Esper um evento representando sua desativação. Esse evento recebe o mesmo *id* de seu evento de ativação correspondente: o valor 1. À medida que João, eventualmente, inicie uma nova ocorrência (*i.e.*, uma nova instância) desse mesmo tipo de situação, ao atributo *id* de seu evento de ativação correspondente será atribuído o valor 2. Dessa forma, é possível relacionar os eventos de ativação e desativação correspondente a cada instância diferente de um mesmo tipo de situação.

SituationDefinition

O componente *SituationDefinition* consiste na classe que agrupa as informações necessárias para a definição de um tipo de situação. Cada tipo de

situação possui um objeto do tipo *SituationDefinition* correspondente, armazenado por uma instância do *SituationManager*. Através dos objetos do tipo *SituationDefinition*, o usuário informa o interesse em iniciar ou interromper a gerência do ciclo de vida de um dado tipo de situação, através do método *start*. A Figura 84 ilustra, de forma simplificada, um trecho do código que implementa a classe *SituationDefinition*.

Os atributos *eplActivate* e *eplDeactivate* (linhas 3, Figura 84) armazenam, respectivamente, as regras EPL (pré-processadas pela plataforma Esper) necessárias para a ativação e desativação do tipo de situação correspondente, definidas previamente nas classes que representam os tipos de situação (contidas no pacote *Situation Class Definitions*). Assim como no componente *Situation* (seção 4.5.1), a associação com seu respectivo tipo de situação ocorre por meio do atributo *sitName* (linha 8, Figura 84). O atributo *listener* (linha 7, Figura 84) mantém uma referencia ao *Listener* que deve ser associado às respectivas regras de ativação e desativação do tipo de situação correspondente. Por padrão, o *Listener* associado corresponde ao *SituationManagerListener* (seção 4.5.4) e não deve ser modificado. O *SituationManagerListener*, juntamente com o *SituationManager*, são responsáveis por controlar o ciclo de vida das situações definidas e armazenadas nos objetos do tipo *SituationDefinition*. O atributo *epAdm* (linha 7, Figura 84) armazena uma referência ao *EPAdministrator*, possibilitando ao objeto do tipo *SituationDefinition* registrar na plataforma Esper as regras de ativação e desativação do tipo de situação correspondente.

Conforme discutido na seção 4.4, o cliente deve invocar o método *start* (linhas 10 a 25, Figura 84) do objeto do tipo *SituationDefinition* para iniciar o gerenciamento do ciclo de vida do tipo de situação associado ao mesmo. Por meio do atributo *epAdmin*, as regras EPL de ativação e desativação do tipo de situação correspondente são registradas na plataforma Esper e são devidamente associadas ao *SituationManagerListener* especificado pelo atributo *listener* (linhas 15 e 16, Figura 84).

Em SIMPLE, sempre que uma regra de ativação ou desativação é acionada, é criado um evento, que corresponde a uma instância do tipo de situação em questão, indicando sua respectiva ativação ou desativação. Esses eventos correspondem,

respectivamente, aos eventos dos tipos *Situation Creation Event* e *Situation Deactivation Event*, definidos no Modelo de Detecção de Situações em CEP (seção 4.1). Durante a primeira invocação do método *start*, para cada tipo de situação, é registrada uma regra EPL responsável por detectar esses eventos correspondentes à ativação e desativação de instâncias do tipo de situação correspondente (linha 22, Figura 84). A regra criada é associada a um *Listener*, de forma que a aplicação cliente possa reagir à detecção dos mesmos. Por padrão, o *Listener* associado a essa regra é representado pelo *SituationListener* (seção 4.5.4). Se necessário, o usuário pode implementar seu próprio *Listener* e substituí-lo pelo *SituationListener*, passando uma referencia do *Listener* implementado ao invocar o método *start*.

```

1 public class SituationDefinition {
2
3     EPPreparedStatement prepared_a, prepared_d;
4     String ID_A, ID_D, ID_S;
5     int numberOfInstances;
6     StatementAwareUpdateListener listener;
7     EPAdministrator epAdm;
8     String sitName;
9
10    public void start(String objectsA[], String objectsD[], StatementAwareUpdateListener SAUL){
11
12        ID_A = this.sitName + this.numberOfInstances + "_a";
13        ID_D = this.sitName + this.numberOfInstances + "_d";
14
15        epAdm.create(prepared_a, ID_A).addListener(listener);
16        epAdm.create(prepared_d, ID_D).addListener(listener);
17
18        if(SAUL == null) SAUL = new SAUL();
19
20        if(numberOfInstances == 0){
21            ID_S = sitName + this.numberOfInstances;
22            epAdm.createEPL("select * from " + sitName, ID_S ).addListener(SAUL);
23        }
24        numberOfInstances++;
25    }
26
27    public void start(){ this.start(null, null, null); }
28    public void start(StatementAwareUpdateListener SAUL){ this.start(null, null, SAUL); }
29    public void start(String objectsA[], String objectsD[]){ this.start(objectsA, objectsD, null); }
30 }

```

Figura 84. Componente *SituationDefinition*

SituationManager

O componente *SituationManager*, ilustrado de forma simplificada pela Figura 85, é responsável por armazenar um objeto do tipo *SituationDefinition* (seção 4.5.2) para cada tipo de situação definido no pacote *Situation Class Definition*, de forma que o usuário possa manifestar seu interesse em iniciar ou interromper o gerenciamento do ciclo de vida dos mesmos. Além disso, esse componente armazena o último evento informando a ativação ou desativação de cada instância

de cada tipo de situação definido (*i.e.*, os últimos *Situation Creation Event* e *Situation Deactivation Event*). Conforme discutido na seção 4.1, através dessas instâncias armazenadas, a plataforma SIMPLE, por meio do componente *SituationManagerListener*, é capaz de obter as informações necessárias para gerir corretamente o ciclo de vida de cada tipo de situação.

O atributo *situationDefinitions* (linha 4, Figura 85) consiste em uma estrutura *hash* responsável por armazenar todos os objetos do tipo *SituationDefinition*. O usuário deve definir todos os tipos de situações modelados a partir do método *start*, sendo efetuada uma chamada desse método para cada tipo distinto. Assim que invocado, o método *start* (linhas 6 a 14, Figura 85) cria um objeto do tipo *SituationDefinition* (linha 7, Figura 85) e o armazena na *hash situationDefinitions* (linha 7, Figura 85). Por sua vez, o atributo *situations* (linha 2, Figura 85) consiste em uma estrutura *hash* responsável armazenar as instâncias dos tipos situações. Deve existir apenas uma instância criada do tipo *SituationManager* por domínio.

```

1 public class SituationManager {
2
3     private static HashMap<String, Situation> situations = null;
4     private static HashMap<String, SituationDefinition> situationDefinitions = null;
5
6     public void start(EPAdministrator epAdm, Situation sit, StatementAwareUpdateListener listener){
7         SituationDefinition sitDef = new SituationDefinition(
8                                     sit.getEplA(),
9                                     sit.getEplD(),
10                                    sit.getSitName(),
11                                    epAdm,
12                                    listener);
13         situationDefinitions.put(sit.getSitName(), sitDef);
14     }
15 }

```

Figura 85. Componente *SituationManager*

SituationListener

O componente *SituationListener* consiste no *Listener* padrão utilizado pelo SIMPLE para monitorar os eventos de ativação e desativação dos tipos de situação especificados. A Figura 86 ilustra, de forma simplificada, a implementação do componente *SituationListener*.

```

1 public class SituationListener implements StatementAwareUpdateListener {
2     public void update(EventBean[] newEvents, EventBean[] oldEvents,
3                       EPStatement stmt, EPServiceProvider epService) {
4
5         Situation sit = (Situation)newEvents[0].getUnderlying();
6         if(sit.isActivated()){
7             sit.doActionAtActivation(newEvents[0]);
8         }else{
9             sit.doActionAtDeactivation(newEvents[0]);
10        }
11    }
12 }
13 }

```

Figura 86. Componente *SituationListener*

Conforme discutido no capítulo 3, a plataforma Esper especifica que todo *Listener* deve implementar o método *update* (linhas 2 a 12, Figura 86), invocado quando um evento (ou um conjunto de eventos) satisfaz as restrições especificadas na *querie* EPL associada ao *Listener*. A implementação do método *update* para o *SituationListener* apenas verifica se o tipo de situação encontra-se ativado (linha 6, Figura 86). Em caso positivo, é invocado o método *doActionAtActivation* correspondente (linha 7, Figura 86). Em caso negativo, é invocado o método *doActionAtDeactivation* correspondente (linha 9, Figura 86).

Esses métodos, por padrão, apenas exibem na tela uma mensagem de ativação, informando o tipo de situação associado ao mesmo, bem como seu identificador único (ID) e seus respectivos *start_time* e *end_time*. Esses métodos podem ser sobrecarregados pelas classes que representam os tipos de situação, permitindo uma ação diferente para cada. Caso a sobrecarga não seja suficiente, o usuário pode implementar seu próprio *Listener* e passar como parâmetro ao método *start* do objeto do tipo *SituationDefinition* correspondente.

SituationManagerListener

O componente *SituationManagerListener* consiste no *Listener* responsável pela gerência do ciclo de vida dos tipos de situações. É de sua responsabilidade criar os respectivos eventos de ativação e desativação (*i.e.*, eventos do tipo *Situation Creation Event* e *Situation Deactivation Event*) referente a cada instância de um tipo

de situação, e enviá-los à plataforma Esper. A Figura 87 ilustra, de forma simplificada, a implementação do método *update* desse componente.

```

1 public void update(EventBean[] Events, EventBean[] oldEvents, EPStatement stmt, epSProvider epS) {
2     for(EventBean event : Events){
3         String stmtName = stmt.getName();
4         String[] stmtNameParts = stmtName.split("_");
5         Boolean isActiveStmt = (stmtNameParts[stmtNameParts.length-1].equals("a"));
6         String key = generateEventKey(stmtName, event);
7         Situation sit = SituationManager.getSituationAt(key);
8
9         if( (isActiveStmt && (sit == null || !sit.isActivated())) ){
10             GenericType sit_event = null;
11             sit_event = typeArgumentClass.newInstance();
12             sit_event = (GenericType) sit_event.createNewSit(event);
13             sit_event.setActivated(true);
14             sit_event.setStart_time(new Date());
15             sit_event.setId(id); id++;
16             sit_event.setKey("@"+id+"S");
17             sit_event.doActionAtCreateActivationEvent(Events);
18             SituationManager.setSituationAt(key, sit_event);
19             epS.getEPRuntime().sendEvent(sit_event);
20         }else{
21             if( !isActiveStmt && sit != null && sit.isActivated()){
22                 GenericType old_event = (GenericType) sit;
23                 GenericType sit_event = (GenericType) old_event.doActionAtCreateDeactivationEvent();
24                 sit_event.setActivated(false);
25                 sit_event.setStart_time(old_event.getStart_time());
26                 sit_event.setEnd_time(new Date());
27                 sit_event.setId(old_event.getId());
28                 sit_event.setKey(old_event.getKey());
29                 SituationManager.setSituationAt(key, sit_event);
30                 epS.getEPRuntime().sendEvent(sit_event);
31             }
32         }
33     }
34 }

```

Figura 87. Componente *SituationManagerListener*

A invocação do método *update* indica que um evento (ou conjunto de eventos) tornou verdadeira as restrições especificadas na *query* EPL associada ao *Listener*. Dessa forma, inicialmente é verificada a *query* EPL corresponde a uma de ativação ou desativação (linha 5, Figura 87). Isso corresponde a verificar se o evento que levou a ativação da mesma corresponde a um evento do tipo *Situation Creation Trigger Event* ou *Situation Deactivation Trigger Event*, conforme especificados no Modelo de Detecção de Situações em CEP (seção 4.1). Essa verificação é realizada através do nome associado à query que foi ativada (linha 3, Figura 87). *Queries* correspondentes à regras de ativação possuem seu nome terminado em “_a”, ao passo que *queries* correspondentes às regras de desativação possuem seu nome terminado em “_d”.

Após isso, deve-se solicitar ao *SituationManager* (seção 4.5.3) a última instância (caso exista) do tipo de situação associado a essa query, por meio do método *getSituationAt* (linha 7, Figura 87). Como as instâncias são armazenadas numa estrutura *hash*, é necessário calcular a chave utilizada para armazenar a instância correspondente (linha 6, Figura 87). A chave utilizada na estrutura *hash* é formada pela concatenação entre o nome da *query* correspondente e os atributos *key* das entidades participantes, especificados na cláusula *select* da *query*. O atributo *activated* da última instância de um tipo de situação indica o *status* do tipo de situação (*i.e.*, *true* para ativado ou *false* para desativado).

De posse do tipo de *query* que levou a invocação do método *update* (*i.e.*, se a *query* corresponde a uma regra ativação ou desativação) e do *status* a última instância do tipo de situação correspondente, é possível inferir se deve ser criado um evento correspondente a uma ativação ou desativação de uma instância de um tipo de situação, ou seja: a criação de um evento correspondente a um (i) *Situation Creation Event*; ou um (i) *Situation Deactivation Event*.

Um evento correspondente a um *Situation Creation Event* deve ser criado caso a *query* corresponda a uma regra de ativação e a última instância do tipo de situação encontra-se desativada (linha 9, Figura 87). Por sua vez, um evento correspondente a um *Situation Deactivation Event* deve ser criado caso a *query* corresponda à uma regra de desativação e a última instância do tipo de situação encontra-se ativada (linha 21, Figura 87).

Caso a primeira possibilidade seja avaliada como verdadeira é criado um novo objeto do tipo de situação correspondente, através do método *createNewSit* (linha 12, Figura 87) do componente *Situation* (seção 4.5.1). Esse objeto criado tem seus atributos *activated*, *start_time*, *id* e *key* inicializados (linhas 13 a 16, Figura 87). O respectivo método *doActionAtCreateActivationEvent* é invocado (linha 17, Figura 87), caso o cliente tenha sobrecarregado o mesmo na classe Java correspondente ao tipo de situação em questão. O objeto criado é, então, armazenado como a última instância do respectivo tipo de situação, através do método *setSituationAt* do *SituationManager* (linha 18, Figura 87). Por fim, o mesmo é enviado à plataforma Esper (linha 18, Figura 87), por meio do método *sendEvent* da interface *EPRuntime* (conforme discutido na seção 3.5).

De forma análoga, caso a segunda possibilidade seja avaliada como verdadeira, o processo é repetido, com algumas diferenças: (i) a criação do objeto do tipo de situação correspondente é realizada através do método *doActionAtCreateDeactivationEvent* (linha 23, Figura 87); e (ii) os atributos *start_time*, *id* e *key* recebem o mesmo valor atribuído ao evento de ativação correspondente, enquanto o atributo *end_time* recebe um novo valor (linhas 24 a 28, Figura 87). Por fim, novamente, o objeto é armazenado como última instância do tipo de situação (linha 29, Figura 87) e enviado à plataforma Esper (linha 30, Figura 87)

Caso nenhuma das duas condições seja satisfeita, é possível concluir que a *query* em questão, necessariamente, indica (i) a ativação de um tipo de situação que já encontra-se ativado ou (ii) a desativação de um tipo de situação que já encontra-se desativado. Em ambos os casos, tal informação faz-se irrelevante ao usuário, levando ao término do método *update* sem a criação de um novo evento.

ANEXO B – CÓDIGO UTILIZADO NA TRANSFORMAÇÃO DE MODELOS SML EM CÓDIGO SIMPLE

Este anexo exhibe o código Aceleo utilizado para transformar instâncias de modelos SML em código a ser executado diretamente pela plataforma SIMPLE, seguindo os padrões apresentados e discutidos no capítulo 5.

```
[comment encoding = UTF-8 /]
[module entity('http://www.example.org/ctx', 'http://www.example.org/sml')]
[import org::eclipse::acceleo::module::sample::main::resources /]
[import org::eclipse::acceleo::module::sample::main::generate /]

[comment template that generates all java code/]
[template public javaEntities(aSMLModel : SMLModel)]
[comment @main/]
[for (n : EntityClass | eAllContents(EntityParticipant).isOfType->asSet()-
>union(contextModel.eAllContents(ctx:EntityClass)->asSet()))]
[javaEntitie(n)/]
[/for]
[for (r : RelatorClass | eAllContents(RelatorParticipant).isOfType->asSet())]
[javaRelator(r)/]
[/for]
[for (s : SituationType | elements.oclAsType(SituationType))]
[javaSituation(s)/]
[/for]
[/template]

[comment datatype representation in java code/]
[template public datatype(d : DataType) post (trim())][d.name.clean()/][[/template]

[comment generate a java class based on a EntityClass/]
[template public javaEntitie(n : EntityClass)]
[file
(getProperty('genDir')+'/' +getProperty('javaDir')+'/' +getProperty('javaPackage').r
eplaceAll('\.', '/')+'model/'+n.name.toUpperFirst()+'.java', false, 'UTF-8')]
package [getProperty('javaPackage')]/.model;
import [getProperty('builtinsPackage')]/.*;

public[if isAbstract] abstract[/if] class [n.name.toUpperFirst().clean()/][if (not
superclass.oclIsUndefined())] extends
[superclass.name.toUpperFirst().clean()/][if] {

    [for (a : Attribute | attribute)]
    private [datatype(a.datatype)/] [name.clean()/];
    [/for]
    [if isAbstract = false]private String key;[/if]

    [for (a : Attribute | attribute)]
    public void set[name.toUpperFirst().clean()/]([datatype(a.datatype)/]
[name.clean()/]) {
        this.[name.clean()/] = [name.clean()/];
    }
}
```

```

    public [datatype(a.datatype)] get[name.toUpperFirst().clean()/( )] {
        return [name.clean()/( )];
    }

    [/for]
    [if isAbstract = false]
    public String getKey() {
        return key;
    }

    public void setKey(String key) {
        this.key = key;
    }
    [/if]
}
[/file]
[/template]

[comment generate a java class based on a EntityClass/]
[template public javaRelator(n : RelatorClass)]
[file
(getProperty('genDir')+'/' +getProperty('javaDir')+'/' +getProperty('javaPackage').r
eplaceAll('\.', '/')+'model/'+n.name.toUpperFirst()+'.java', false, 'UTF-8')]
package [getProperty('javaPackage')/( )].model;
import [getProperty('builtinsPackage')/( )].*;

public class [n.name.toUpperFirst().clean()/( )] extends Situation{

    [for (a : Attribute | attribute)]
    private [datatype(a.datatype)] [name.clean()/( )];
    [/for]
    [for (a: Association | self.associations())]
    private [a.target.name.toUpperFirst().clean()/( )] [name.clean()/( )];
    [/for]
    private String key;

    [for (a : Attribute | attribute)]
    public [datatype(a.datatype)] get[name.toUpperFirst().clean()/( )] {
        return [name.clean()/( )];
    }

    public void set[name.toUpperFirst().clean()/( )]([datatype(a.datatype)]
[name.clean()/( )]) {
        this.[name.clean()/( )] = [name.clean()/( )];
    }

    [/for]
    [for (a: Association | self.associations())]
    public [a.target.name.toUpperFirst().clean()/( )]
get[name.toUpperFirst().clean()/( )] {
        return [name.clean()/( )];
    }

    public void
set[name.toUpperFirst()/( )]([a.target.name.toUpperFirst().clean()/( )] [name.clean()/( )])
{
        this.[name.clean()/( )] = [name.clean()/( )];
    }
}

```

```

    [//for]
    public String getKey() {
        return key;
    }

    public void setKey(String key) {
        this.key = key;
    }

    @Override
    public Situation createNewSit(EventBean event) {
        // TODO Auto-generated method stub
        return null;
    }

    @Override
    public Object doActionAtCreateDeactivationEvent() {
        // TODO Auto-generated method stub
        return null;
    }
}
[/file]

[/template]

[template public javaSituation(s : SituationType)]
[file
(getProperty('genDir')+'/' +getProperty('javaDir')+'/' +getProperty('javaPackage')).r
eplaceAll('\\.', '/')+'/' +situation/' +s.name()+'.java', false, 'UTF-8')]
package [getProperty('javaPackage')/].situation;

import com.espertech.esper.client.EventBean;

import [getProperty('javaPackage')/].model.*;
import [getProperty('javaPackage')/].situation.*;
import [getProperty('builtinsPackage')/].*;

public class [s.name()/] extends Situation {

    [for (p : Participant | s.elements->filter(Participant))]
    [javaSituation_attribute_declaration(p)/]
    [/for]

    [for (p : Participant | s.elements->filter(Participant))]
    [javaSituation_attribute_get_set(p)/]
    [/for]

    @Override
    public [s.name()/]() {
        setSitName("[s.name.toUpperFirst()/]");

        setEplA("[epl_ativacao(s)/]");

        setEplA("[epl_desativacao(s)/]");
    }

    @Override

```

```

    public Situation createNewSit(EventBean event) {
        [s.name.toUpperFirst()] [s.name.toLowerFirst() /] = new [s.name/]();

        try{
            [for (p : Participant | s.elements->filter(Participant))]
            [javaSituation_createNewSet(p)]
            [/for]
        }catch(Exception e){
            System.out.println("[s.name.toUpperFirst()]: " + e);
        }

        return [s.name.toLowerFirst() /];
    }

    @Override
    public Object doActionAtCreateDeactivationEvent() {
        [s.name.toUpperFirst()] [s.name.toLowerFirst() /] = new [s.name/]();

        try{
            [for (p : Participant | s.elements->filter(Participant))]
            [javaSituation_doActionAtCreateDeactivationEvent(p)]
            [/for]
        }catch(Exception e){
            System.out.println("[s.name.toUpperFirst()]: " + e);
        }

        return [s.name.toLowerFirst() /];
    }
}

[/file]
[/template]

[template public javaSituation_attribute_declaration(p : Participant)
post(trim())]
private [print_participant_name(p)] [binding_name(p)];
[/template]

[template public javaSituation_attribute_get_set(p : Participant) post(trim())]
[javaSituation_attribute_get_set_1(p)]
[/template]

[template public javaSituation_attribute_get_set_1(p : Participant) post(trim())]
public void set[javaSituation_attribute_get_set_name(p)](
[print_participant_name(p)] [binding_name(p)]) {
    this.[binding_name(p)] = [binding_name(p)];
}

public [print_participant_name(p)]
get[javaSituation_attribute_get_set_name(p)]() {
    return [binding_name(p)];
}
[/template]

[template public javaSituation_attribute_get_set_name(p : Participant)
post(trim())]
[if(p.nodeParameter.oclIsUndefined())]
[print_participant_name(p) /][index_name(p) /]

```

```

[else]
[p.nodeParameter.name.toUpperFirst() /]
[/if]
[/template]

[template public javaSituation_attribute_get_set_more_than_1(p : Participant)
post(trim())]
public void set[print_participant_name(p)/](int index,
[print_participant_name(p)/] [print_participant_name_lower(p)/]) {
    this.[print_participant_name_lower(p)/][ '[' /]index[ ']' /] =
[print_participant_name_lower(p)/];
}

public [print_participant_name(p)/] get[print_participant_name(p)/](int index) {
    return [print_participant_name_lower(p)/][ '[' /]index[ ']' /];
}
[/template]

[template public javaSituation_createNewSet(p : Participant) post(trim())]

[p.eContainer(SituationType).name.toLowerFirst()
/]set[javaSituation_attribute_get_set_name(p)/]([print_participant_name(p)/])eve
nt.get("[binding_name(p)/]");
[/template]

[template public javaSituation_doActionAtCreateDeactivationEvent(p : Participant)
post(trim())]
[p.eContainer(SituationType).name.toLowerFirst()
/]set[javaSituation_attribute_get_set_name(p)/](this.get[javaSituation_attribute_
get_set_name(p)/]());
[/template]

[comment encoding = UTF-8 /]
[module generate('http://www.example.org/ctx', 'http://www.example.org/sml')]

[import org::eclipse::acceleo::module::sample::main::resources /]

[template public epl_ativacao(s: SituationType) post (replaceAll('\n',' '))]
select
    [epl_ativacao_select(s)/]
from
    [epl_ativacao_from(s)/]

where
    [epl_ativacao_where(s)/]
[/template]

[template public epl_desativacao(s: SituationType) post (replaceAll('\n',' '))]
select
    [epl_desativacao_select(s)/]
from
    [epl_desativacao_from(s)/]

[epl_desativacao_where(s)/]
[/template]

```

```

[ comment SELECT - ACTIVATE /]

[template public epl_ativacao_select(s: SituationType) ]
[let participants : OrderedSet(Participant) = s.elements->filter(Participant) ]
[select_print_all_participant_names(participants)/]
[/let]
[/template]

[template public select_print_all_participant_names (participants:
Set(Participant)) ]
[for (p: Participant | participants) separator (' ')] [select_fully_qualified_name
(p) /] [/for] [select_key (participants)/]
[/template]

[template public select_fully_qualified_name (p: Participant) post (trim())]
[if (p.eContainer(SituationType).elements->filter(Link)-
>select(entity.oclAsType(EntityParticipant)->size() > 0 and entity = p)-
>first().oclIsUndefined() = false )]
[let link: Link = p.eContainer(SituationType).elements->filter(Link)-
>select(entity.oclAsType(EntityParticipant)->size() > 0 and entity = p)->first()]
[fully_qualified_name(link.relator) /]. [link.isOfType.name /] as
[binding_name(p)/]
[/let]
[elseif (p.targetRelation.source->filter(SituationParameterReference)->size() > 0)
]
[let param: SituationParameterReference =
p.targetRelation.source.oclAsType(SituationParameterReference)->first()]
[fully_qualified_name(param.situation) /]. [param.parameter.name.toLowerFirst() /]
as [binding_name(p)/]
[/let]
[else]
[fully_qualified_name(p) /]
[/if]
[/template]

[template public select_key (participants : Set(Participant)) post (trim())]
[let all_participants : OrderedSet(Participant) = participants->asOrderedSet() ]
[let spr_participants : OrderedSet(Participant) =
participants.eContainer(SituationType).elements-
>filter(SituationParameterReference)->asOrderedSet()->spr_participants() ]
[let link_participants : OrderedSet(Participant) =
participants.eContainer(SituationType).elements->filter(Link)->asOrderedSet()-
>link_participants() ]
[let all_relevant_participants : OrderedSet(Participant) = all_participants-
>osminus(spr_participants)->osminus(link_participants) ]
[for (p: Participant |
all_relevant_participants)] [if(p.oclIsTypeOf(ExistsSituation) = false)],
[binding_name(p) /].key as key[i/][if][if]
[/let]
[/let]
[/let]
[/let]
[/template]

```



```

[ comment FROM - ACTIVATE /]

[template public epl_ativacao_from(s: SituationType) post (trim()) ]
[let participants : OrderedSet(Participant) = s.elements->filter(Participant) ]
[from_print_all_participant_names(participants)/]
[/let]
[/template]

[template public from_print_all_participant_names (participants: Set(Participant))
post (trim().replace('or \\(', '\\(')) ]
[let all_participants : OrderedSet(Participant) = participants->asOrderedSet() ]
[let spr_participants : OrderedSet(Participant) =
participants.eContainer(SituationType).elements-
>filter(SituationParameterReference)->asOrderedSet()->spr_participants() ]
[let link_participants : OrderedSet(Participant) =
participants.eContainer(SituationType).elements->filter(Link)->asOrderedSet()-
>link_participants() ]
[let all_relevant_participants : OrderedSet(Participant) = all_participants-
>osminus(spr_participants)->osminus(link_participants) ]
[if(all_relevant_participants->size() = 1)]
[from_1_participant (all_relevant_participants->first()) /]
[else]
pattern['['/] every (
    [from_more_than_1_participant (all_relevant_participants->asSequence(),
1)/]
)
['['/]
[/if]
[/let]
[/let]
[/let]
[/let]
[/let]
[/template]

[template public from_1_participant (p: Participant) post (trim())]
[print_participant_name(p) /][from_window_1_participant(p)/] as [binding_name(p)
/]
[/template]

[template public from_window_1_participant (p: Participant) post (trim())]
[if (p.oclIsTypeOf(ExistsSituation) = false) ]
    [if(p.reference->size() > 0)]
        [for (at : AttributeReference | p.reference) ]
            [if (at.attribute.name = 'final time') ]
                [for( s: ComparativeRelation | at.sourceRelation ) ]
                    [if(s.relation.name = 'within the past')]

                .win:time([s.target.oclAsType(Literal).value/])

            [/if]
        [/for]
    [/if]
[/for]
[/if]
[/template]

[template public from_window_more_than_1_participant (p: Participant,
participants: Sequence(Participant)) post (trim())]

```

```

[if(participants->indexOf(p) = participants->size()) ]
[if (p.oclIsTypeOf(ExistsSituation) = false) ]
    [if(p.reference->size() > 0)]
        [for (at : AttributeReference | p.reference) ]
            [if (at.attribute.name = 'start time' or at.attribute.name =
'final time') ]
                [for( s: ComparativeRelation | at.sourceRelation ) ]
                    [if(s.relation.name = 'within the past')]
                        where
timer:within([s.target.oclAsType(Literal).value/])
                    [/if]
                [/for]
            [/if]
        [/for]
    [/if]
[/if]
[/if]
[/template]

[template public from_more_than_1_participant (participants:
Sequence(Participant), k: Integer) post (trim())]
[if (participants->size() = k)]
or [from_more_than_1_participant_combination(participants)/]
[else]
    [for (p:Participant | participants)]
        [if( (i >= k) and (i < (participants->size()+1)) )]
[from_more_than_1_participant(ExchangeCharacters(participants,k, i), k + 1)/]
        [/if]
    [/for]
[/if]
[/template]

[template public from_more_than_1_participant_combination (participants:
Sequence(Participant)) post (trim())]
[for(pFixo:Participant | participants)separator (' -> ')]
(
    every [from_combination_print_participant_fixo(pFixo)/]
    [if(participants->indexOf(pFixo) > 1)]
        [for(pVar:Participant | participants)]
            [if(participants->indexOf(pVar) < participants-
>indexOf(pFixo))]
                [from_combination_print_participant_var(pVar)/]
            [/if]
        [/for]
    )[from_window_more_than_1_participant(pFixo, participants)/]
[/if]
[/for]
)
[/template]

[template public from_combination_print_participant_fixo(p: Participant)]
[binding_name(p)/] =
[print_participant_name(p)/][if(p.oclIsTypeOf(EntityParticipant) =
false)]([/if][from_print_participant_status(p,
true)/][if(p.oclIsTypeOf(EntityParticipant) = false)])[/if]
[/template]

[template public from_combination_print_participant_var(p: Participant)]

```

```

[if(p.ocIsTypeOf(EntityParticipant) = false)]and not
[print_participant_name(p)/]([from_print_participant_status(p,
false)/][from_print_participant_id(p)/])[if]
[/template]

[template public from_print_participant_status(p: Participant, type: Boolean)]
[if(p.ocIsKindOf(SituationParticipant))]activated =
[participant_status(p.ocAsType(SituationParticipant),
type)/][elseif(p.ocIsKindOf(RelatorParticipant))]activated = [type/][if]
[/template]

[template public from_print_participant_id(p: Participant)]
[if(p.ocIsTypeOf(SituationParticipant))], id =
[binding_name(p)/].id[elseif(p.ocIsTypeOf(RelatorParticipant))], key =
[binding_name(p)/].key[/if]
[/template]

[template public participant_status(p: SituationParticipant, type: Boolean) post
(trim())]
[if(type = false)][if(p.isPast = false)]false[else]true[/if]
[else][if(p.isPast = false)]true[else]false[/if][if]
[/template]

[ comment WHERE - ACTIVATE /]

[template public epl_ativacao_where(s: SituationType) post (trim()) ]
[let participants : OrderedSet(Participant) = s.elements->filter(Participant) ]
[where_print_all_participant_names(participants)/]
[activate_where_all_functions (s) /]
[/let]
[/template]

[template public where_print_all_participant_names (participants:
Set(Participant)) post (trim().replaceAll('\n', ' and ')) ]
[let relators : OrderedSet(RelatorParticipant) =
participants.eContainer(SituationType).elements->filter(RelatorParticipant)-
>select(nodeParameter.ocIsUndefined()->asOrderedSet())]
[if(relators->size() = 1)][binding_name(relators->first()) /].activated =
true[/if]
[/let]
[for(p : Participant | participants)][relations(p, true)/][for]
[/template]

[template public relations(node : Node, activation_rule : Boolean)]
[if
(node.ocIsTypeOf(EntityParticipant))][formal_relations(node.ocAsType(EntityParti
cipant), activation_rule)/][elseif
(node.ocIsTypeOf(RelatorParticipant))][formal_relations(node.ocAsType(RelatorPar
ticipant), activation_rule)/][elseif
(node.ocIsKindOf(SituationParticipant))][situation_relations(node.ocAsType(Situa
tionParticipant), activation_rule)/][if ]
[/template]

[template public formal_relations(p : Participant, activation_rule : Boolean) post
(trim())]
[for (rel : ComparativeRelation | p.sourceRelation)]
[comparative_relation(p, rel, rel.target, activation_rule) /]
[/for ]
[formal_relations_attributes(p, activation_rule)/]

```

```
[/template]
```

```
[template public formal_relations_attributes(p : Participant, activation_rule :
Boolean) post (trim())]
[for (att : AttributeReference | p.reference)]
[for (rel : ComparativeRelation | sourceRelation)]
[comparative_relation(att,rel,rel.target, activation_rule)]
[/for ]
[/for ]
[/template]
```

```
[template public comparative_relation(source: Node, rel : ComparativeRelation,
target: Node, activation_rule : Boolean) post(trim())]
[if(activation_rule = true)]
[if(rel.relation.name = 'equals' or rel.relation.name = 'greater than' or
rel.relation.name = 'less than')][if (rel.isNegated)]not ([/if
][comparative_relation_primitive(source, rel, target, activation_rule) ][if
(rel.isNegated)]][/if ]
[else][if (rel.isNegated)]not ([/if ][comparative_relation_user(source, rel,
target, activation_rule) ][if (rel.isNegated)]][/if ][/if]
[else]
[if(rel.relation.name = 'equals' or rel.relation.name = 'greater than' or
rel.relation.name = 'less
than')][deactivate_comparative_relation_primitive(source, rel, target) ]/
[else][deactivate_comparative_relation_user(source, rel, target) ][/if]
[/if]
[/template]
```

```
[template public comparative_relation_primitive(source: Node, rel :
ComparativeRelation, target: Node, activation_rule : Boolean) post(trim())]
[if(source.ocIsTypeOf(SituationParameterReference))][print_situation_type_name_re
lation(source.ocAsType(SituationParameterReference).situation,
activation_rule)][value(source.ocAsType(SituationParameterReference).situation)/
].[source.ocAsType(SituationParameterReference).parameter.name.clean()/]
[print_relation(rel.relation)] [print_situation_type_name_relation(target,
activation_rule)][value(target)]/
[else][print_situation_type_name_relation(source,
activation_rule)][value(source)]/
[print_relation(rel.relation)][parameter_primitive(rel)]/
[print_situation_type_name_relation(target,
activation_rule)][value(target)]/[if]
[/template]
```

```
[template public comparative_relation_user(source: Node, rel :
ComparativeRelation, target: Node, activation_rule : Boolean) post(trim())]
[if(source.ocIsTypeOf(SituationParameterReference))][print_relation(rel.relation)
/][[print_situation_type_name_relation(source.ocAsType(SituationParameterReferenc
e).situation,
activation_rule)][value(source.ocAsType(SituationParameterReference).situation)/
].[source.ocAsType(SituationParameterReference).parameter.name.clean()/],
[print_situation_type_name_relation(target, activation_rule)][value(target)]/
[parameter_user(rel)]/)]
[else][print_relation(rel.relation)]([print_situation_type_name_relation(source,
activation_rule)][value(source)]/, [print_situation_type_name_relation(target,
activation_rule)][value(target)]/ [parameter_user(rel)]/)]/[if]
[/template]
```

```

[template public parameter_primitive(rel: ComparativeRelation) ]
[if (not parameter.ocIsUndefined() and not (parameter = ''))]
['('/')'[parameter/]['(')'/][if ]
[/template]

[template public parameter_user(rel: ComparativeRelation) ]
[if (not parameter.ocIsUndefined() and not (parameter = ''))], '[parameter/]'[/if
]
[/template]

[template public situation_relations(s : SituationParticipant, activation_rule :
Boolean)]
[let all_participants : OrderedSet(Participant) =
s.eContainer(SituationType).elements->filter(Participant)->asOrderedSet() ]
[let spr_participants : OrderedSet(Participant) =
s.eContainer(SituationType).elements->filter(SituationParameterReference) -
>asOrderedSet()->spr_participants() ]
[let link_participants : OrderedSet(Participant) =
s.eContainer(SituationType).elements->filter(Link)->asOrderedSet()-
>link_participants() ]
[let all_relevant_participants : OrderedSet(Participant) = all_participants-
>osminus(spr_participants)->osminus(link_participants) ]
[if(all_relevant_participants->size() = 1)]
[fully_qualified_name(s)/].activated = [if(s.isPast)]false[else]true[/if]
[/if]
[/let]
[/let]
[/let]
[/let]
[for (p : SituationParameterReference | parameter)]
[for (r : ComparativeRelation | p.sourceRelation)]
[if(r.target.ocIsKindOf(SituationParameterReference) = true)]
[comparative_relation(p,r,r.target, activation_rule)/]
[/if]
[/for ]
[/for ]
[/template]

[template public print_situation_type_name_relation(n: Node, activation_rule :
Boolean) post(trim())]
[if(n.ocIsTypeOf(Literal) = false)]
[if(activation_rule = false)][print_situation_type_name(n)/].[/if]
[/if]
[/template]

[template public value(n : Node) post (trim())]
[if (n.ocIsTypeOf(EntityParticipant))]
[fully_qualified_name(n.ocAsType(EntityParticipant))/]
[elseif (n.ocIsTypeOf(RelatorParticipant))]
[fully_qualified_name(n.ocAsType(RelatorParticipant))/]
[elseif (n.ocIsTypeOf(AttributeReference))]
[attr_name(n.ocAsType(AttributeReference))/]
[elseif (n.ocIsTypeOf(Literal))]
[literal(n.ocAsType(Literal))/]
[elseif (n.ocIsTypeOf(SituationParameterReference))]
[sit_param_name(n.ocAsType(SituationParameterReference))/]
[elseif (n.ocIsTypeOf(SituationParticipant))]
[fully_qualified_name(n.ocAsType(SituationParticipant))/]
[/if ]

```

```

[/template]

[template public attr_name(attr : AttributeReference)]
[fully_qualified_name(attr.entity)/].[attribute.name.clean()/]
[/template]

[template public sit_param_name(param : SituationParameterReference)]
[fully_qualified_name(param.situation)/].[param.parameter.name.clean()/]
[/template]

[template public literal(literal : Literal)]
[if (literal.dataType.name = 'String')]
'[literal.value/]'
[else]
[literal.value/]
[/if ]
[/template]

[template public activate_where_all_functions (s: SituationType) post(trim()) ]
[let all_functions : OrderedSet(Function) = s.elements->filter(Function)]
[let param : OrderedSet(Parameter) = all_parameters(all_functions)]
[let internal_functions : OrderedSet(Node) = parameter_functions(param)]
[let external_functions : OrderedSet(Node) =
osminus(all_functions,internal_functions)]
[for(n: Node | external_functions)]
[if(n.ocIsTypeOf(Function))]
and [activate_where_function_main(n.ocAsType(Function))/]
[/if]
[/for]
[/let]
[/let]
[/let]
[/let]
[/let]
[/template]

[template public activate_where_function_main (function_node: Node) post(trim()) ]
[if(function_node.ocIsTypeOf(Function))]
[let f: Function = function_node.ocAsType(Function)]
[let param : OrderedSet(Parameter) = parameters(f)]
[f.function.name/]( [activate_where_function_print_all_parameters (param)/])
[/let]
[/let]
[/if]
[/template]

[template public activate_where_function_print_all_parameters (parameters:
OrderedSet(Parameter)) post(trim().replaceAll('\n', '')) ]
[for(p: Parameter | parameters) separator(', ')]
[let n: Node = p.value]
[if(n.ocIsTypeOf(Function))][activate_where_function_main(n)/][else][activate_whe
re_function_print_parameter_name(n)/][endif]
[/let]
[/for]
[/template]
[template public activate_where_function_print_parameter_name (n: Node)
post(trim()) ]
[deactivate_value(n)/]
[/template]

```

```
[ comment SELECT - DEACTIVATE /]
```

```
[template public epl_desativacao_select(s: SituationType) ]
[let participants : OrderedSet(Participant) = s.elements->filter(Participant)-
>select(p:Participant| p.oclIsTypeOf(ExistsSituation) = false) ]
[deactivate_select_print_all_participant_names(participants)/]
[/let]
[/template]
```

```
[template public deactivate_select_print_all_participant_names (participants:
Set(Participant)) ]
[participants->asOrderedSet()->first().eContainer(SituationType).name/][for (p:
Participant | participants) ][/for][deactivate_select_key (participants) /]
[/template]
```

```
[template public deactivate_select_key (participants : Set(Participant)) post
(trim())]
[let all_participants : OrderedSet(Participant) = participants->asOrderedSet() ]
[let spr_participants : OrderedSet(Participant) =
participants.eContainer(SituationType).elements-
>filter(SituationParameterReference)->asOrderedSet()->spr_participants() ]
[let link_participants : OrderedSet(Participant) =
participants.eContainer(SituationType).elements->filter(Link)->asOrderedSet()-
>link_participants() ]
[let all_relevant_participants : OrderedSet(Participant) = all_participants-
>osminus(spr_participants)->osminus(link_participants) ]
[for (p: Participant | all_relevant_participants)],
[p.eContainer(SituationType).name/].[binding_name(p) /].key as key[i/][/for]
[/let]
[/let]
[/let]
[/let]
[/template]
```

```
[comment FROM - DEACTIVATE /]
```

```
[template public epl_desativacao_from(s: SituationType) post (trim()) ]
[let participants : OrderedSet(Participant) = s.elements->filter(Participant) ]
[if(participants->select(p:Participant| p.oclIsTypeOf(ExistsSituation) = true)-
>size() > 0 )]
[deactivate_from_print_all_participant_names_exists(participants)/]
[else]
[deactivate_from_print_all_participant_names(participants)/]
[/if]
[/let]
[/template]
```

```
[template public deactivate_from_print_all_participant_names_exists (participants:
Set(Participant)) post (trim()) ]
[let existParticipant : ExistsSituation = participants->filter(ExistsSituation)-
>asOrderedSet()->first()]
[let situationType : SituationType = participants->asOrderedSet()-
>first().eContainer(SituationType) ]
pattern['[/]
    [situationType.name/] = [situationType.name/] ->
    every (
        timer:interval( [deactivate_from_exists_value(existParticipant) /]
    )
```

```

        and not [print_participant_name(existParticipant)/]
([from_print_participant_status(existParticipant, true)/])
    )
    [' ']/]
[/let]
[/let]
[/template]

[template public deactivate_from_exists_value (p: ExistsSituation) post (trim()) ]
[if(p.reference->size() > 0)]
[for (at : AttributeReference | p.reference) ]
[if (at.attribute.name = 'final time' or at.attribute.name = 'initial time') ]
[for( s: ComparativeRelation | at.sourceRelation ) ]
[if(s.relation.name = 'within the past')]
[s.target.oclAsType(Literal).value/]
[/if]
[/for]
[/if]
[/for]
[/if]
[/template]

[template public deactivate_from_print_all_participant_names (participants:
Set(Participant)) post (trim()) ]
[participants->asOrderedSet()-
>first().eContainer(SituationType).name/].std:unique(id) as [participants-
>asOrderedSet()->first().eContainer(SituationType).name/][for (p: Participant |
participants) ][deactivate_from_print_participant_name (p) /][for]
[/template]

[template public deactivate_from_print_participant_name (p: Participant) post
(trim()) ]
[if( (p.eContainer(SituationType).elements->filter(Link)->select(entity = p)-
>size() = 0) and p.oclIsTypeOf(ExistsSituation) = false)]
    [if((p.targetRelation.source->filter(SituationParameterReference)->size() =
0) )]
        [deactivate_from_print_class_name (p)/]
    [/if]
[/if]
[/template]

[template public deactivate_from_print_class_name (p: Participant) post (trim()) ]
[let unnamed : OrderedSet(Participant) = (p.eContainer(SituationType).elements-
>filter(Participant))]
[if(p.oclIsTypeOf(EntityParticipant))]
[if( (unnamed->filter(EntityParticipant)->select(e:EntityParticipant| e.isOfType =
p.oclAsType(EntityParticipant).isOfType)->indexOf(p)) = 1 )]
, [print_participant_name(p)/].std:unique(key) as [deactivate_from_binding_name
(p)/]
[/if]
[/if]
[if(p.oclIsTypeOf(RelatorParticipant))]
[if( (unnamed->filter(RelatorParticipant)->select(e:RelatorParticipant| e.isOfType
= p.oclAsType(RelatorParticipant).isOfType)->indexOf(p)) = 1 )]
, [print_participant_name(p)/].std:unique(key) as [deactivate_from_binding_name
(p)/]
[/if]
[/if]
[if(p.oclIsTypeOf(SituationParticipant))]

```



```

[if( (named->filter(SituationParticipant)-
>select(e:SituationParticipant|e.situationType =
p.oclAsType(SituationParticipant).situationType->indexOf(p)) = 1 )]
, [print_participant_name(p)/].std:lastevent() as [deactivate_from_binding_name
(p)/]
[/if]
[/if]
[/let]
[/template]

[template public deactivate_from_binding_name (p: Participant) post (trim()) ]
[print_participant_name_lower(p)/][index_name(p) /]
[/template]

[ comment WHERE - DEACTIVATE /]

[template public epl_desativacao_where(s: SituationType) ]
[let participants : OrderedSet(Participant) = s.elements->filter(Participant) ]
[if(participants->select(p:Participant| p.oclIsTypeOf(ExistsSituation) = true)-
>size() > 0 )]
    [if(participants->select(p:Participant| p.sourceRelation.oclIsUndefined() =
false)->size() > 0)]
where
[for(p : Participant | participants) ][relations(p, false)/][for]
[/if]
[else]
where
    [deactivate_where_print_all_participant_names(participants)/]
[/if]
[deactivate_where_all_functions (s) /]
[/let]
[/template]

[template public deactivate_where_print_all_participant_names (participants:
Set(Participant)) post(replaceAll('or not \\(\\)', '').replaceAll('\\( or',
'\\(').replaceAll('\\n', ' ').replaceAll('\\t', ' ').replaceAll(' ', '
').replaceAll('or or', 'or').replaceAll('\\) or \\(', '\\) \\nor
\\(').replaceAll('\\) or \\(', '\\) \\nor \\(')) ]
[print_situation_type_name(participants)/].activated is true
and ( [deactivate_where_print_all_participant_names_iteration(participants) /]
[if(participants->asOrderedSet()->first().eContainer(SituationType).elements-
>filter(Function)->size() = 0 )])[if]
[/template]

[template public deactivate_where_print_all_participant_names_iteration
(participants: Set(Participant)) post(trim()) ]
[for (p: Participant | participants) ][deactivate_where_print_participant_name (p,
i)/][for]
[for(p : Participant | participants) ][relations(p, false)/][for]
[/template]

[template public deactivate_where_print_comparation (p: Participant) post(trim())
]
[if(p.oclIsTypeOf(SituationParticipant)) ]
[deactivate_where_print_comparation_id(p)/]
[else]
[deactivate_where_print_comparation_key(p)/]
[/if]
[/template]

```

```

[template public deactivate_where_print_comparation_key (p: Participant)
post(trim()) ]
[print_situation_type_name(p)/].[binding_name(p)/].key =
[deactivate_where_binding_name_1(p)/].key
[/template]

[template public deactivate_where_print_comparation_id (p: Participant)
post(trim()) ]
[print_situation_type_name(p)/].[binding_name(p)/].id =
[deactivate_where_binding_name_1(p)/].id
[/template]

[template public deactivate_where_print_comparation_key (n: Node) post(trim()) ]
[if (n.ocIsTypeOf(EntityParticipant)) ]
[print_situation_type_name(n)/].[fully_qualified_name(n.ocAsType(EntityParticipan
t)/)].key = [deactivate_value(n.ocAsType(EntityParticipant)/)].key
[elseif (n.ocIsTypeOf(RelatorParticipant)) ]
[print_situation_type_name(n)/].[fully_qualified_name(n.ocAsType(RelatorParticipa
nt)/)].key = [deactivate_value(n.ocAsType(RelatorParticipant)/)].key
[elseif (n.ocIsTypeOf(SituationParticipant)) ]
[print_situation_type_name(n)/].[fully_qualified_name(n.ocAsType(SituationPartici
pant)/)].key = [deactivate_value(n.ocAsType(SituationParticipant)/)].key
[elseif (n.ocIsTypeOf(AttributeReference)) ]
[print_situation_type_name(n)/].[fully_qualified_name(n.ocAsType(AttributeReferen
ce).entity)/].key =
[deactivate_value(n.ocAsType(AttributeReference).entity)/].key
[/if ]
[/template]

[template public checkFirstOr (index1: Integer, index2: Integer) ]
[if( (index1 > 1) or (index2 >1))] or [/if]
[/template]

[template public checkRelevantParticipantsOr (p: Node) ]
[let unamed : OrderedSet(Participant) = (p.eContainer(SituationType).elements-
>filter(Participant))]
[let all_participants : OrderedSet(Participant) =
p.eContainer(SituationType).elements->filter(Participant)->asOrderedSet() ]
[let spr_participants : OrderedSet(Participant) =
p.eContainer(SituationType).elements->filter(SituationParameterReference)-
>asOrderedSet()->spr_participants() ]
[let link_participants : OrderedSet(Participant) =
p.eContainer(SituationType).elements->filter(Link)->asOrderedSet()-
>link_participants() ]
[let all_relevant_participants : OrderedSet(Participant) = all_participants-
>osminus(spr_participants)->osminus(link_participants) ]
[if(all_relevant_participants->filter(RelatorParticipant)->asSet()->size() > 0 or
all_relevant_participants->filter(SituationParticipant)->asSet()->size() > 0)] or
[/if]
[/let]
[/let]
[/let]
[/let]
[/let]
[/template]

[template public deactivate_where_print_participant_name (p: Participant, index:
Integer) ]

```

```

[if( (p.eContainer(SituationType).elements->filter(Link)->select(entity = p)-
>size() = 0) and p.ocIsTypeOf(ExistsSituation) = false)]
    [if((p.targetRelation.source->filter(SituationParameterReference)->size() =
0) )]
        [let unnamed : OrderedSet(Participant) =
(p.eContainer(SituationType).elements->filter(Participant))]
        [let all_participants : OrderedSet(Participant) =
p.eContainer(SituationType).elements->filter(Participant)->asOrderedSet() ]
        [let spr_participants : OrderedSet(Participant) =
p.eContainer(SituationType).elements->filter(SituationParameterReference)-
>asOrderedSet()->spr_participants() ]
        [let link_participants : OrderedSet(Participant) =
p.eContainer(SituationType).elements->filter(Link)->asOrderedSet()-
>link_participants() ]
        [let all_relevant_participants : OrderedSet(Participant) =
all_participants->osminus(spr_participants)->osminus(link_participants) ]
        [if( (p.ocIsTypeOf(EntityParticipant) = true) and
(all_relevant_participants->select(self.ocAsType(EntityParticipant).isOfType.name
= p.ocAsType(EntityParticipant).isOfType.name)->size() > 1) )]
            [checkFirstOr(index, 1)][deactivate_where_print_class_name(p) /]
            [elseif( (p.ocIsTypeOf(RelatorParticipant) = true) and
(all_relevant_participants-
>select(self.ocAsType(RelatorParticipant).isOfType.name =
p.ocAsType(RelatorParticipant).isOfType.name)->size() > 1) )]
                [checkFirstOr(index, 1)][deactivate_where_print_class_name(p) /]
                [elseif( (p.ocIsTypeOf(SituationParticipant) = true) and
(all_relevant_participants-
>select(self.ocAsType(SituationParticipant).situationType.name =
p.ocAsType(SituationParticipant).situationType.name)->size() > 1) )]]
                    [checkFirstOr(index, 1)][deactivate_where_print_class_name(p) /]
                    [else]
                    [checkFirstOr(index, 1)][deactivate_where_print_class_name(p) /]
                    [/if]
                    [/let]
                    [/let]
                    [/let]
                    [/let]
                    [/let]
            [/if]
        [/if]
    [/if]
[/template]

[template public deactivate_where_print_class_name (p: Participant) post (trim())
]
[if(p.ocIsTypeOf(RelatorParticipant))]
([deactivate_where_print_comparation(p)/] and
[deactivate_where_binding_name_1(p)/].activated is not true)
[/if]
[if(p.ocIsTypeOf(SituationParticipant))]
([deactivate_where_print_comparation(p)/] and
[deactivate_where_binding_name_1(p)/].activated is not
[if(p.ocAsType(SituationParticipant).isPast = true)]false[else]true[/if])
[/if]
[/template]

[template public deactivate_where_binding_name (p: Participant) post (trim()) ]
[print_participant_name_lower(p)/][index_name(p) /]

```

```
[/template]
```

```
[template public deactivate_comparative_relation_primitive(source: Node, rel :
ComparativeRelation, target: Node) post(trim())]
[if(source.ocIsTypeOf(SituationParameterReference))]
[checkRelevantParticipantsOr(source) /] ([deactivate_where_print_comparation_id
(source.ocIsType(SituationParameterReference).situation)/] and not
([deactivate_value(source)/] [print_relation(rel.relation)/]
[print_situation_type_name_relation(target, false)/][value(target)/]))
[checkRelevantParticipantsOr(source) /] ([deactivate_where_print_comparation_id
(target.ocIsType(SituationParameterReference).situation)/] and not
([print_situation_type_name_relation(source.ocIsType(SituationParameterReference)
.situation,
false)/][value(source.ocIsType(SituationParameterReference).situation)/].[source.
ocIsType(SituationParameterReference).parameter.name.clean()/]
[print_relation(rel.relation)/] [deactivate_value(target)/]))
[elseif(source.ocIsTypeOf(Literal))]
[checkRelevantParticipantsOr(source) /] ([deactivate_where_print_comparation_key
(target)/] and not ([print_situation_type_name_relation(source,
false)/][value(source)/] [deactivate_value(target)/]))
[elseif(target.ocIsTypeOf(Literal))]
[checkRelevantParticipantsOr(source) /] ([deactivate_where_print_comparation_key
(source)/] and not ([deactivate_value(source)/] [print_relation(rel.relation)/]
[print_situation_type_name_relation(target, false)/][value(target)/]))
[else]
[checkRelevantParticipantsOr(source) /] ([deactivate_where_print_comparation_key
(source)/] and not ([deactivate_value(source)/] [print_relation(rel.relation)/]
[print_situation_type_name_relation(target, false)/][value(target)/]))
[checkRelevantParticipantsOr(source) /] ([deactivate_where_print_comparation_key
(target)/] and not ([print_situation_type_name_relation(source,
false)/][value(source)/] [print_relation(rel.relation)/]
[deactivate_value(target)/]))
[/if]
[/template]
```

```
[template public deactivate_comparative_relation_user(source: Node, rel :
ComparativeRelation, target: Node) post(trim())]
[if(source.ocIsTypeOf(SituationParameterReference))]
[checkRelevantParticipantsOr(source) /] ([deactivate_where_print_comparation_id
(source.ocIsType(SituationParameterReference).situation)/] and [if
(rel.isNegated)]not ([/if ]not
([print_relation(rel.relation)/]([deactivate_value(source.ocIsType(SituationParam
eterReference).situation)/].[source.ocIsType(SituationParameterReference).paramet
er.name.clean()/], [print_situation_type_name_relation(target,
false)/][value(target)/] [parameter_user(rel)/])))[if (rel.isNegated)])[/if ]
[checkRelevantParticipantsOr(source) /] ([deactivate_where_print_comparation_id
(target.ocIsType(SituationParameterReference).situation)/] and [if
(rel.isNegated)]not ([/if ]not
([print_relation(rel.relation)/]([print_situation_type_name_relation(source.ocIsT
ype(SituationParameterReference).situation,
false)/][value(source.ocIsType(SituationParameterReference).situation)/].[source.
ocIsType(SituationParameterReference).parameter.name.clean()/],
[deactivate_value(target)/] [parameter_user(rel)/])))[if (rel.isNegated)])[/if ]
[elseif(source.ocIsTypeOf(Literal))]
[checkRelevantParticipantsOr(source) /] ([deactivate_where_print_comparation_key
(target)/] and [if (rel.isNegated)]not ([/if ]not
([print_relation(rel.relation)/]([print_situation_type_name_relation(source,
false)/][value(source)/], [deactivate_value(target)/] [parameter_user(rel)/])))[if
(rel.isNegated)])[/if ]
```

```

[elseif(target.oclIsTypeOf(Literal))]
[checkRelevantParticipantsOr(source) /] ([deactivate_where_print_comparation_key
(source)/] and [if (rel.isNegated)]not ([/if ]not
([print_relation(rel.relation)/]([deactivate_value(source)/],
[print_situation_type_name_relation(target, false)/][value(target)/]
[parameter_user(rel)/]))[if (rel.isNegated)]][/if ]
[else]
[checkRelevantParticipantsOr(source) /] ([deactivate_where_print_comparation_key
(source)/] and [if (rel.isNegated)]not ([/if ]not
([print_relation(rel.relation)/]([deactivate_value(source)/],
[print_situation_type_name_relation(target, false)/][value(target)/]
[parameter_user(rel)/]))[if (rel.isNegated)]][/if ]
[checkRelevantParticipantsOr(source) /] ([deactivate_where_print_comparation_key
(target)/] and [if (rel.isNegated)]not ([/if ]not
([print_relation(rel.relation)/]([print_situation_type_name_relation(source,
false)/][value(source)/], [deactivate_value(target)/] [parameter_user(rel)/]))[if
(rel.isNegated)]][/if ]
[/if]
[/template]

[template public deactivate_value(n : Node) post (trim())]
[if (n.oclIsTypeOf(EntityParticipant))]
[fully_qualified_name_deactivate_1(n.oclAsType(EntityParticipant))/]
[elseif (n.oclIsTypeOf(RelatorParticipant))]
[fully_qualified_name_deactivate_1(n.oclAsType(RelatorParticipant))/]
[elseif (n.oclIsTypeOf(AttributeReference))]
[deactivate_attr_name(n.oclAsType(AttributeReference))/]
[elseif (n.oclIsTypeOf(Literal))]
[literal(n.oclAsType(Literal))/]
[elseif (n.oclIsTypeOf(SituationParameterReference))]
[deactivate_sit_param_name(n.oclAsType(SituationParameterReference))/]
[elseif (n.oclIsTypeOf(SituationParticipant))]
[fully_qualified_name_deactivate_1(n.oclAsType(SituationParticipant))/]
[/if ]
[/template]

[template public deactivate_attr_name(attr : AttributeReference)]
[fully_qualified_name_deactivate_1(attr.entity)/].[attribute.name.clean()/]
[/template]

[template public deactivate_sit_param_name(param : SituationParameterReference)]
[fully_qualified_name_deactivate_1(param.situation)/].[param.parameter.name.clean(
)/]
[/template]

[template public deactivate_where_all_functions (s: SituationType) post(trim()) ]
[let all_functions : OrderedSet(Function) = s.elements->filter(Function)]
[let param : OrderedSet(Parameter) = all_parameters(all_functions)]
[let internal_functions : OrderedSet(Node) = parameter_functions(param)]
[let external_functions : OrderedSet(Node) =
osminus(all_functions,internal_functions)]
[for(n: Node | external_functions)]
[if(n.oclIsTypeOf(Function))]
[deactivate_where_function_main(n.oclAsType(Function), n.oclAsType(Function)) /]
[/if]
[/for]
[if(s.elements->filter(Function)->size() > 0 )][/if]
[/let]
[/let]

```

```

[/let]
[/let]
[/template]

[template public deactivate_where_function_main (external_function: Function,
current_function: Function) post(trim().replaceAll('\n', ' ').replaceAll('or
or', 'or').replaceAll(' or ', '\nor ').replaceAll('\|' or \|(' , '\|)\nor \|(')) ]
[let param : OrderedSet(Parameter) = parameters(current_function)]
[for(p: Parameter | param)]
[let n: Node = p.value]
[if( n.ocIsKindOf(Participant) or n.ocIsKindOf(AttributeReference) or
n.ocIsKindOf(SituationParameterReference) )]
[checkRelevantParticipantsOr(n)] or
([deactivate_where_function_main_comparation(n)] and not(
[deactivate_where_function_main_function_parameters(external_function, n)]) )
[elseif(n.ocIsKindOf(Function))]
[deactivate_where_function_main(external_function, n.ocAsType(Function))]/]
[/if]
[/let]
[/for]
[/let]
[/template]

[template public deactivate_where_function_main_comparation (n: Node) post(trim())
]
[if(n.ocIsTypeOf(SituationParameterReference))]
[deactivate_where_print_comparation_id
(n.ocAsType(SituationParameterReference).situation)]/]
[elseif(n.ocIsTypeOf(SituationParticipant))]
[deactivate_where_print_comparation_id (n.ocAsType(SituationParticipant))]/]
[elseif(n.ocIsTypeOf(AttributeReference))]
[deactivate_where_print_comparation_key (n.ocAsType(AttributeReference).entity)]/]
[elseif(n.ocIsTypeOf(Participant))]
[deactivate_where_print_comparation_key (n.ocAsType(Participant))]/]
[/if]
[/template]

[template public deactivate_where_function_main_function_parameters
(function_node: Node, current_node: Node) post(trim()) ]
[if(function_node.ocIsTypeOf(Function))]
[let f: Function = function_node.ocAsType(Function)]
[let param : OrderedSet(Parameter) = parameters(f)]
[f.function.name]/]([deactivate_where_function_print_all_parameters (param,
current_node)/])
[/let]
[/let]
[/if]
[/template]

[template public deactivate_where_function_print_all_parameters (parameters:
OrderedSet(Parameter), current_node: Node) post(trim()) ]
[for(p: Parameter | parameters) separator(', ')]
[let n: Node = p.value]
[if(n.ocIsTypeOf(Function))][deactivate_where_function_main_function_parameters(n
, current_node)]/]
[else][deactivate_where_function_print_parameter_name(n, current_node)]/[/if]
[/let]
[/for]
[/template]

```

```

[template public deactivate_where_function_print_parameter_name (n: Node,
current_node: Node) post(trim()) ]
[let all_nodes : OrderedSet(Node) = n.eContainer(SituationType).elements-
>filter(Node)]
[if(all_nodes->indexOf(n) = all_nodes-
>indexOf(current_node))][deactivate_value(n)/]
[else][print_situation_type_name_relation(n, false)/][value(n)/][endif]
[/let]
[/template]

comment encoding = UTF-8 /]
[**
 * This module contains some helpers to code generation
 */]
[module resources('http://www.example.org/sml', 'http://www.example.org/ctx' )]

[query public add_dep(acc:OrderedSet(Participant), e : Participant) :
OrderedSet(Participant) =
    if not e.oclIsUndefined() then
        if acc->includes(e) then
            acc
        else
            acc->osunion(dependency(e))->append(e)
        endif
    else
        acc
    endif
/]

[query public add_function(acc:OrderedSet(Function), e : Function) :
OrderedSet(Function) =
    if not e.oclIsUndefined() then
        if acc->includes(e) then
            acc
        else
            acc->append(e)
        endif
    else
        acc
    endif
/]

[query public participant(e : SituationTypeElement) : Participant =
    if e.oclIsKindOf(Participant) then
        e.oclAsType(Participant)
    else
        if e.oclIsTypeOf(AttributeReference) then
            e.oclAsType(AttributeReference).entity
        else
            if e.oclIsTypeOf(SituationParameterReference) then
                e.oclAsType(SituationParameterReference).situation
            else
                null
            endif
        endif
    endif
/]

```

```

[query public participant_formal_dependency(e : Participant) :
OrderedSet(Participant) =
    e.sourceRelation->iterate(
        r:ComparativeRelation;
        acc : OrderedSet(Participant) = OrderedSet{}
        | acc->add_dep(r.target.participant())
    )
/]

[query public attribute_formal_dependency(e : Participant) :
OrderedSet(Participant) =
    e.reference.sourceRelation->iterate(
        r:ComparativeRelation;
        acc : OrderedSet(Participant) = OrderedSet{}
        | acc->add_dep(r.target.participant())
    )
/]

[query public situation_formal_dependency(e : SituationParticipant) :
OrderedSet(Participant) =
    e.parameter.sourceRelation->iterate(
        r:ComparativeRelation;
        acc : OrderedSet(Participant) = OrderedSet{}
        | acc->add_dep(r.target.participant())
    )
/]

[query public relational_dependency(e : RelatorParticipant) :
OrderedSet(Participant) =
    e.links->iterate(
        l: Link;
        acc : OrderedSet(Participant) = OrderedSet{}
        | acc->add_dep(l.entity)
    )
/]

[query public dependency(e : Participant) : OrderedSet(Participant) =
    if e.oclIsTypeOf(RelatorParticipant) then
        relational_dependency(e.oclAsType(RelatorParticipant))-
>osunion(attribute_formal_dependency(e.oclAsType(RelatorParticipant)))
    else
        if e.oclIsTypeOf(EntityParticipant) then
            participant_formal_dependency(e.oclAsType(EntityParticipant))-
>osunion(attribute_formal_dependency(e.oclAsType(EntityParticipant)))
        else
            if e.oclIsKindOf(SituationParticipant) then

                participant_formal_dependency(e.oclAsType(SituationParticipant))-
>osunion(situation_formal_dependency(e.oclAsType(SituationParticipant)))
            else
                OrderedSet(SituationTypeElement){}
            endif
        endif
    endif
/]

```



```

[query public ordered(origin : OrderedSet(Participant)) : OrderedSet(Participant)
=
    origin->iterate(
        e : Participant;
        acc : OrderedSet(Participant) = OrderedSet{}
        | acc->add_dep(e)
    )
/]

[template public name(n : OclAny) post (trim())]
[if (n.ocIsTypeOf(SituationType))]
    [n.ocAsType(SituationType).name.toUpperFirst().clean()/]
[/if]
[if (n.ocIsTypeOf(EntityParticipant))]

    [n.ocAsType(EntityParticipant).isOfType.name.toUpperFirst().clean()/]
[/if]
[if (n.ocIsTypeOf(RelatorParticipant))]

    [n.ocAsType(RelatorParticipant).isOfType.name.toUpperFirst().clean()/]
[/if]
[/template]

[query public time_attrubute(a : AttributeReference) : Literal =
a.sourceRelation.target->filter(Literal)-
>select(l:Literal|l.dataType.name='Time')->first() /]

[template public time_litral(l:Literal) post (trim())]
    [l.value.replaceAll(' ', '').replaceAll('days|day','d')/]
[/template]

[query public parameters(f : Function) : OrderedSet(Parameter) =
    f.parameter->iterate(
        q:Parameter;
        acc : OrderedSet(Parameter) = OrderedSet{}
        | acc->append(q)
    )
/]

[query public all_parameters(nodes : OrderedSet(Node)) : OrderedSet(Parameter) =
    nodes->iterate(
        n:Node;
        acc : OrderedSet(Parameter) = OrderedSet{}
        | if(n.ocIsTypeOf(Function)) then
            acc->osunion(parameters(n.ocAsType(Function)))
        else
            acc
        endif
    )
/]

[query public parameter_functions(param : OrderedSet(Parameter)) :
OrderedSet(Node) =
    param->iterate(
        p:Parameter;
        acc : OrderedSet(Node) = OrderedSet{}
        | if p.value.ocIsKindOf(Function) then
            acc->append(p.value)
        else

```

```

        acc
    endif
)
/]

[query public osunion(la: OrderedSet(Parameter), lb: OrderedSet(Parameter)):
OrderedSet(Parameter) =
    lb->iterate(
        ei : Parameter;
        acci : OrderedSet(Parameter) = la
        | acci->append(ei)
    )
/]

[query public osminus(la: OrderedSet(Node), lb: OrderedSet(Node)):
OrderedSet(Node) =
    la->iterate(
        ei : Node;
        acci : OrderedSet(Node) = OrderedSet{}
        | if lb->includes(ei) then
            acci
        else
            acci->append(ei)
        endif
    )
/]

[query public superclasses(s : Set(EntityClass)) : Set(EntityClass) =
self->iterate(
    e : EntityClass;
    acc : Set(EntityClass) = Set(EntityClass){}
    | if acc->includes(e) then
        acc
    else
        if e.superclass.oclIsUndefined() then
            acc->including(e)
        else
            acc->including(e)-
>union(superclasses(Set{e.superclass}))
        endif
    endif
)
/]

[query public associations(p : ModelClass) : OrderedSet(Association) =
eContainer(ContextModel).elements->filter(Association)-
>select(a:Association|a.source = p or a.target = p) /]

[query public osminus(la: OrderedSet(Participant), lb: OrderedSet(Participant)):
OrderedSet(Participant) =
    la->iterate(
        ei : Participant;
        acci : OrderedSet(Participant) = OrderedSet{}
        | if lb->includes(ei) then
            acci
        else
            acci->append(ei)
        endif
    )
/]

```

```

/]

[query public osunion(la: OrderedSet(Participant), lb: OrderedSet(Participant)):
OrderedSet(Participant) =
    lb->iterate(
        ei : Participant;
        acci : OrderedSet(Participant) = la
        | if acci->includes(ei) then
            acci
        else
            acci->append(ei)
        endif
    )
/]

[query public link_participants(la: OrderedSet(Link)): OrderedSet(Participant) =
    la->iterate(
        ei : Link;
        acci : OrderedSet(Participant) = OrderedSet{}
        | acci->append(ei.entity)
    )
/]

[query public spr_participants(nodes: OrderedSet(SituationParameterReference)):
OrderedSet(Participant) =
    nodes->iterate(
        n : SituationParameterReference;
        acci : OrderedSet(Participant) = OrderedSet{}
        | acci->osunion(spr_sourceRelations(n))
    )
/]

[query public spr_sourceRelations(n: Node): OrderedSet(Participant) =
    n.sourceRelation->iterate(
        r : ComparativeRelation;
        acci : OrderedSet(Participant) = OrderedSet{}
        | acci->append(check_node_participant(r.target))
    )
/]

[query public check_node_participant (n: Node): Participant =
    if(n.ocIsKindOf(Participant)) then
        n.ocAsType(Participant)
    else
        null
    endif
/]

[query public ExchangeCharacters(acc:Sequence(Participant), p1:Integer, p2:Integer
) : Sequence(Participant) =
    acc->iterate(
        p: Participant;
        result: Sequence(Participant) = Sequence{}
        | if(acc->indexOf(p) = p1) then
            result->append(acc->at(p2))
        else
            if(acc->indexOf(p) = p2) then
                result->append(acc->at(p1))
            else

```

```

                                result->append(p)
                            endif
                        endif
                    )
/]

[template public print_relation(cr: ComparativeFormalRelation) post (trim())]
[if (cr.oclIsUndefined())]
=
[else ]
[if (name = 'equals')]
=
[elseif (name = 'greater than')]
>
[elseif (name = 'less than')]
<
[else ]
[cr.name.clean()]
[/if ]
[/if ]
[/template]

[template public print_situation_type_name(p: Node) post (trim())]
[p.eContainer(SituationType).name/]
[/template]

[template public print_situation_type_name(participants: Set(Participant)) post
(trim())]
[participants->asOrderedSet()->first().eContainer(SituationType).name/]
[/template]

[template public binding_name (p: Participant) post (trim())]
[if(p.nodeParameter.oclIsUndefined() = false)]
[p.nodeParameter.name/]
[else][print_participant_name_lower(p) /][index_name(p) /][[/if]
[/template]

[template public binding_name_antigo (p: Participant) post (trim())]
[if(p.nodeParameter.oclIsUndefined())]
[print_participant_name_lower(p) /][index_name(p) /]
[else]
[p.nodeParameter.name /]
[/if]
[/template]

[template public print_participant_name (p: Participant) post (trim())]
[if (p.oclIsTypeOf(EntityParticipant))]
    [p.oclAsType(EntityParticipant).isOfType.name.toUpperFirst().clean()/]
[elseif (p.oclIsTypeOf(RelatorParticipant))]
    [p.oclAsType(RelatorParticipant).isOfType.name.toUpperFirst().clean()/]
[elseif (p.oclIsKindOf(SituationParticipant))]
    [p.oclAsType(SituationParticipant).situationType.name.toUpperFirst().clean(
)/]
[/if ]
[/template]

[template public print_participant_name_lower (p: Participant) post (trim())]

```

```

[if (p.ocIsTypeOf(EntityParticipant))]
    [p.ocAsType(EntityParticipant).isOfType.name.toLowerFirst().clean()/]
[elseif (p.ocIsTypeOf(RelatorParticipant))]
    [p.ocAsType(RelatorParticipant).isOfType.name.toLowerFirst().clean()/]
[elseif (p.ocIsKindOf(SituationParticipant))]
    [p.ocAsType(SituationParticipant).situationType.name.toLowerFirst().clean(
)/]
[/if ]
[/template]

[template public print_parameter_name (p: Parameter) post (trim())]
[if (p.value.ocIsTypeOf(EntityParticipant))]
[p.value.ocAsType(EntityParticipant).isOfType.name/]
[elseif (p.value.ocIsTypeOf(RelatorParticipant))]
[p.value.ocAsType(RelatorParticipant).isOfType.name/]
[elseif (p.value.ocIsTypeOf(AttributeReference))]
[p.value.ocAsType(AttributeReference).attribute.name/]
[elseif (p.value.ocIsTypeOf(Literal))]
[p.value.ocAsType(Literal).value/]
[elseif (p.value.ocIsTypeOf(SituationParameterReference))]
[p.value.ocAsType(SituationParameterReference).situation.situationType.name/]
[elseif (p.value.ocIsTypeOf(SituationParticipant))]
[p.value.ocAsType(SituationParticipant).situationType.name/]
[elseif (p.value.ocIsTypeOf(Function))]
[p.value.ocAsType(Function).function.name/]
[/if ]
[/template]

[template public index_name(n : ExportableNode) post (trim())]
[let unamed : OrderedSet(Participant) = (n.eContainer(SituationType).elements-
>filter(Participant))]
[if (n.ocIsTypeOf(EntityParticipant))]
[unamed->filter(EntityParticipant)->select(e:EntityParticipant| e.isOfType =
n.ocAsType(EntityParticipant).isOfType)->indexOf(n)/]
[elseif (n.ocIsTypeOf(RelatorParticipant))]
[unamed->filter(RelatorParticipant)->select(e:RelatorParticipant| e.isOfType =
n.ocAsType(RelatorParticipant).isOfType)->indexOf(n)/]
[elseif (n.ocIsKindOf(SituationParticipant))]
[unamed->filter(SituationParticipant)-
>select(e:SituationParticipant|e.situationType =
n.ocAsType(SituationParticipant).situationType)->indexOf(n)/]
[/if ]
[/let ]
[/template]

[template public index_name_zero(n : ExportableNode) post (trim())]
[let unamed : OrderedSet(Participant) = (n.eContainer(SituationType).elements-
>filter(Participant))]
[if (n.ocIsTypeOf(EntityParticipant))]
[unamed->filter(EntityParticipant)->select(e:EntityParticipant| e.isOfType =
n.ocAsType(EntityParticipant).isOfType)->indexOf(n)-1/]
[elseif (n.ocIsTypeOf(RelatorParticipant))]
[unamed->filter(RelatorParticipant)->select(e:RelatorParticipant| e.isOfType =
n.ocAsType(RelatorParticipant).isOfType)->indexOf(n)-1/]
[elseif (n.ocIsKindOf(SituationParticipant))]
[unamed->filter(SituationParticipant)-
>select(e:SituationParticipant|e.situationType =
n.ocAsType(SituationParticipant).situationType)->indexOf(n)-1/]
[/if ]

```

```

[/let ]
[/template]

[template public fully_qualified_name (p: Participant) post (trim())]
  [if (p.eContainer(SituationType).elements->filter(Link)-
>select(entity.oclAsType(EntityParticipant)->size() > 0 and entity = p)-
>first().oclIsUndefined() = false )]
    [let link: Link = p.eContainer(SituationType).elements->filter(Link)-
>select(entity.oclAsType(EntityParticipant)->size() > 0 and entity = p)->first()]
    [fully_qualified_name(link.relator) /].[link.isOfType.name /]
  [/let]
  [elseif (p.targetRelation.source->filter(SituationParameterReference)->size() >
0) ]
    [let param: SituationParameterReference =
p.targetRelation.source.oclAsType(SituationParameterReference)->first()]
    [fully_qualified_name(param.situation) /].[param.parameter.name.toLowerFirst() /]
  [/let]
[else]
[binding_name(p) /]
[/if]
[/template]

[template public deactivate_where_binding_name_1 (p: Participant) post (trim()) ]
[print_participant_name_lower(p)/]1
[/template]

[template public fully_qualified_name_deactivate_1 (p: Participant) post (trim())]
  [if (p.eContainer(SituationType).elements->filter(Link)-
>select(entity.oclAsType(EntityParticipant)->size() > 0 and entity = p)-
>first().oclIsUndefined() = false )]
    [let link: Link = p.eContainer(SituationType).elements->filter(Link)-
>select(entity.oclAsType(EntityParticipant)->size() > 0 and entity = p)->first()]
    [fully_qualified_name_deactivate_1(link.relator) /].[link.isOfType.name /]
  [/let]
  [elseif (p.targetRelation.source->filter(SituationParameterReference)->size() >
0) ]
    [let param: SituationParameterReference =
p.targetRelation.source.oclAsType(SituationParameterReference)->first()]
    [fully_qualified_name_deactivate_1(param.situation)
/] .[param.parameter.name.toLowerFirst() /]
  [/let]
[else]
[deactivate_where_binding_name_1(p) /]
[/if]
[/template]

[template public clean(name : String) post (trim())]
[name.replaceAll('\\s+', '_')/]
[/template]

```