RODRIGO HERNANDEZ SOARES

Gerenciamento Dinâmico de Modelos de Contexto: Estudo de Caso Baseado em CEP

Dissertação apresentada ao Programa de Pós–Graduação do Instituto de Informática da Universidade Federal de Goiás, como requisito parcial para obtenção do título de Mestre em Ciência da Computação.

Área de concentração: Sistemas Distribuídos.

Orientador: Prof. Ricardo Couto Antunes da Rocha

Goiânia
2012

RODRIGO HERNANDEZ SOARES

Gerenciamento Dinâmico de Modelos de Contexto: Estudo de Caso Baseado em CEP

Dissertação defendida no Programa de Pós–Graduação do Instituto de Informática da Universidade Federal de Goiás como requisito parcial para obtenção do título de Mestre em Ciência da Computação, aprovada em 29 de Maio de 2012, pela Banca Examinadora constituída pelos professores:

Prof. Ricardo Couto Antunes da Rocha
Instituto de Informática – UFG
Presidente da Banca

Profa. Patrícia Dockhorn Costa
Departamento de Informática – UFES

Prof. Renato de Freitas Bulcão Neto
Instituto de Informática – UFG

Rodrigo Hernandez Soares

Graduou–se em Ciências da Computação pela UFG - Universidade Federal de Goiás. Durante sua graduação, foi estagiário do CERCOMP/UFG, atuando como membro da equipe de segurança. Além disso, participou como voluntário de projetos de pesquisa para desenvolvimento de serviços de *middleware* para sistemas baseados em localização. Durante o mestrado, foi bolsista REUNI, contribuindo, como monitor, com as disciplinas oferecidas pelo Instituto de Informática que lidam predominantemente com tópicos de segurança da informação.

Aos meus pais.

Agradecimentos

Em primeiro lugar, quero agradecer aos meus pais, que, com todo carinho e dedicação que um pai e uma mãe podem oferecer a um filho, me ensinaram tudo que eu precisava saber para chegar onde cheguei.

Quero agradecer também a todos os meus familiares, por sempre estarem dispostos a oferecer o suporte que preciso para seguir vivendo. Em especial, quero agradecer meu irmão, Rafael, pela amizade do dia a dia.

Agradeço também à minha namorada, Thais, pelos momentos de ternura em épocas de tanta pressão.

Um agradecimento especial aos meus amigos, por compreenderem os momentos de ausência.

Quero expressar a minha gratidão ao meu orientador, professor Ricardo, pelo tempo, paciência e sabedoria oferecidos a mim.

Gostaria de agradecer também a todos os professores que contribuiram com a minha formação. Em especial, ao professor Vagner José do Sacramento Rodrigues, que foi um grande incentivador da minha caminhada rumo ao mestrado.

Por fim, meus sinceros agradecimentos ao pessoal da coordenação de pós-graduação do INF e ao pessoal da secretaria. Em especial, Edir e Ricardo, que sempre estiveram disponíveis para me auxiliar, com toda presteza, nos assuntos extra-aula que devem ser tratados ao longo do mestrado.

Context is always as relevant as concept.

Terry Olson,
Focus on Faculty, Vol 15(2), Brigham Young University Faculty Center.

Resumo

Soares, Rodrigo Hernandez. **Gerenciamento Dinâmico de Modelos de Contexto: Estudo de Caso Baseado em CEP**. Goiânia, 2012. 70p. Dissertação de Mestrado. Instituto de Informática, Universidade Federal de Goiás.

Modelos contextuais que descrevem cenários de computação sensível ao contexto dinâmicos normalmente precisam ser frequentemente atualizados. Alguns exemplos de situações que motivam essas atualizações são o surgimento de novos serviços e provedores de informações contextuais, a mobilidade das entidades descritas nesses modelos, dentre outros. Normalmente, atualizações em modelos implicam em redesenvolvimento dos componentes arquiteturais dos sistemas sensíveis ao contexto baseados nesses modelos. Porém, como em cenários dinâmicos essas atualizações tendem a ser mais frequentes, é desejável que elas ocorram em tempo de execução.

Essa dissertação apresenta uma infraestrutura para gerenciamento dinâmico de modelos de contexto baseada nos fundamentos de processamento complexo de eventos, ou CEP. Essa infraestrutura permite que as abstrações fundamentais a partir das quais um modelo é construído sejam atualizadas em tempo de execução. Como essas atualizações podem causar impactos nos sistemas baseados nos modelos atualizados, essa dissertação identifica e analisa esses impactos, os quais são reproduzidos em um estudo de caso que tem como finalidade avaliar a infraestrutura proposta através da demonstração de como ela lida com os impactos mencionados.

Palavras–chave

Computação Sensível ao Contexto, Modelos de Contexto, Gerenciamento Dinâmico, Atualizações em tempo de execução.

Abstract

Soares, Rodrigo Hernandez. **Dynamic Management of Context Models: A Case Study Based on CEP**. Goiânia, 2012. 70p. MSc. Dissertation. Instituto de Informática, Universidade Federal de Goiás.

Context models that describe dynamic context-aware scenarios usually need to be frequently updated. Some examples of situations that motivate these updates are the appearance of new services and context providers, the mobility of the entities described in these models, among others.

Generally, updates on models imply redevelopment of the architectural components of context-aware systems based on these models. However, as these updates in dynamic scenarios tend to be more frequent, it is desirable that they occur at runtime.

This dissertation presents an infrastructure for dynamic management of context models based on the fundamentals of complex event processing, or CEP. This infrastructure allows the fundamental abstractions from which a model is built to be updated at runtime. As these updates can impact systems based on the updated models, this dissertation identifies and analyzes these impacts, which are reproduced in a case study that aims to evaluate the proposed infrastructure by demonstrating how it deals with the impacts mentioned.

Keywords

Context-aware computing, Context models, Dynamic Management, Updates at runtime.

Sumário

Lista de Figuras **11**

Lista de Tabelas **12**

1 Introdução **13**
- 1.1 Objetivos e Contribuições 14
- 1.2 Metodologia 14
- 1.3 Organização da Dissertação 15

2 Gerenciamento Dinâmico de Modelos de Contexto **16**
- 2.1 Modelos de Contexto 16
- 2.2 Arquiteturas de Sistemas Sensíveis ao Contexto 18
- 2.3 Cenário de Manutenção Dinâmica de Modelos de Contexto 20
- 2.4 Atualização Suave em Modelos de Contexto 21
- 2.5 Componentes de Modelo de Contexto 24
- 2.6 Impactos do Dinamismo em Componentes de Modelo 25
- 2.7 Sumário 27

3 Processamento de Fluxos de Informação Utilizando a Tecnologia CEP/Esper **29**
- 3.1 Esper e EPL 29
- 3.2 Componentes de Modelo no Esper 30
  - 3.2.1 Eventos e Entidades 30
  - 3.2.2 Regras 31
  - 3.2.3 Operadores 32
- 3.3 Sumário 33

4 Infraestrutura para Gerenciamento Dinâmico de Modelos de Contexto **34**
- 4.1 Requisitos 34
- 4.2 Gerenciamento de Modelos de Contexto 35
- 4.3 Arquitetura e Implementação 39
  - 4.3.1 Camada de Gerenciamento de Modelos de Contexto 40
  - 4.3.2 Camada de Processamento de Eventos 43
  - 4.3.3 Camada de Gerenciamento de Eventos de Entrada 47
  - 4.3.4 Camada de Gerenciamento de Eventos de Saída 48
- 4.4 Discussão 48
- 4.5 Sumário 49

5 Estudo de Caso **50**
5.1 Descrição da Aplicação 50
5.2 Primeiro Cenário 51
5.3 Segundo Cenário 54
5.4 Implementação 56
5.4.1 Implementação do primeiro cenário 56
5.4.2 Implementação do Segundo Cenário 58
5.4.3 Implementação dos modelos como *bundles* OSGi 58
5.5 Discussão 61
5.6 Limitações 62
5.7 Sumário 62

6 Trabalhos Relacionados **63**

7 Conclusões **65**
7.1 Contribuições 66
7.2 Trabalhos Futuros 66

Referências Bibliográficas **68**

Lista de Figuras

2.1 Arquitetura em camadas do sistema sensível ao contexto proposto por Henricksen & Indulska. 18
2.2 Modelo de funcionamento dos IFPS adaptado para o universo da computação sensível ao contexto. 19
2.3 Inconsistência causada por uma atualização em um modelo de contexto utilizado por uma aplicação. 23
(b) Atualização realizada sobre um conceito do modelo utilizado pela aplicação. 23

4.1 Organização hierárquica dos modelos de contexto. 36
4.2 Ciclo de vida de um modelo de contexto. 37
4.3 Operações de escrita executadas sobre um modelo de contexto utilizado por uma aplicação. 38
(b) Operação de remoção executada. 38
(c) Operação de alteração executada. 38
4.4 Arquitetura da infraestrutura de gerenciamento dinâmico de modelos proposta. 40
4.5 Ciclo de vida de um *bundle* OSGi. 43
4.6 Criação e uso de operadores de semântica variável no Esper. 45
4.7 Exemplo de uso de um *proxy*. 45
4.8 Criação e uso de operadores de semântica variável no Esper com uso de *proxy*. 46

5.1 Protótipo da aplicação de chat sensível ao contexto. 50
5.2 Cenário inicial de uso da aplicação de *chat* do departamento de informática. 51
5.3 Novo cenário com modelo de contexto atualizado. 55
5.4 Diagrama de produção e consumo de eventos referente ao primeiro cenário. 57
5.5 Diagrama de produção e consumo de eventos referente ao segundo cenário. 59

Lista de Tabelas

2.1 Relação entre atualização em componentes de modelo, impactos causados e agentes afetados. 27

4.1 Cabeçalhos adicionais de um *bundle* que o caracterizam como um modelo de contexto. 42

4.2 Estados equivalentes dos ciclos de vida de *bundles* e modelos. 43

CAPÍTULO 1

Introdução

O principal objetivo da computação sensível ao contexto é permitir o desenvolvimento de aplicações capazes de adaptar os seus comportamentos através do uso de informações de contexto. Segundo Dey [8], contexto é qualquer informação que pode ser utilizada para caracterizar a situação de entidades, que são pessoas, lugares ou objetos considerados relevantes para a interação entre um usuário e uma aplicação.

Uma das primeiras etapas do desenvolvimento de aplicações sensíveis ao contexto consiste na especificação das informações contextuais que serão utilizadas por essas aplicações [9]. Esse processo é conhecido como modelagem contextual, e o resultado final, chamado de modelo de contexto, é utilizado para descrever o cenário sensível ao contexto que deverá ser implementado. Existem diferentes abordagens para modelagem de contexto [2], as quais se diferem por aspectos como expressividade, formalismo, aplicabilidade a diferentes domínios, dentre outros aspectos. Exemplos dessas abordagens são ontologias, atributo-valor e orientadas a objetos.

Um problema independente da abordagem utilizada mas que tem relação com modelos de contexto é que, para a descrição de cenários dinâmicos, é possível que as estruturas desses modelos precisem ser frequentemente atualizadas. Considere, por exemplo, o contexto *Ocupado*, que descreve o estado de indisponibilidade de uma pessoa. Esse contexto pode depender de diversos fatores, como por exemplo a localização da pessoa, o estado do ambiente em que a mesma se encontra (luminosidade, nível de ruídos sonoros), a relação dessa pessoa com outras que se encontram ao seu redor, dentre outros. O problema é que muitos fatores que influenciam a indisponibilidade de uma pessoa podem passar a ser relevantes, para efeito de inferência do contexto *Ocupado*, apenas quando o modelo de contexto associado ao cenário já estiver pronto e sendo utilizado por uma aplicação que tenha interesse por esse contexto.

Normalmente, atualizações em modelos de contexto implicam no redesenvolvimento das implementações correspondentes a esses modelos. Em cenários dinâmicos, é desejável que essas atualizações ocorram sem que os componentes arquiteturais de um sistema sensível ao contexto baseado nos modelos em questão precisem ser redesenvolvidos ou reimplantados, já que essas atualizações tendem a ser mais frequentes nesses

cenários. Por isso, esse trabalho propõe que atualizações em modelos de contexto devem ser "percebidas" em tempo de execução pelos componentes arquiteturais de um sistema sensível ao contexto.

Essa dissertação trata do gerenciamento dinâmico de modelos de contexto, que envolve a atualização, em tempo de execução, de elementos que compõem um modelo, com a finalidade de adaptá-lo a situações de caráter dinâmico e imprevisível. Para tanto, é preciso definir os tipos de atualização e os elementos que compõem um modelo aos quais essas atualizações serão aplicadas.

As atualizações tratadas nessa dissertação são chamadas de *atualizações suaves*, e elas são aplicadas a elementos chamados de *componentes de modelo*, que são as abstrações fundamentais a partir das quais um modelo é construído. A principal característica das atualizações suaves é que elas não exigem o redesenvolvimento de um sistema e nem causam inconsistências no mesmo. Porém, mesmo atualizações suaves podem provocar impactos em sistemas sensíveis ao contexto, e essa dissertação apresenta uma infraestrutura para gerenciamento dinâmico de modelos de contexto que implementa mecanismos responsáveis por tratar os impactos gerados por atualizações suaves em componentes de modelo.

1.1 Objetivos e Contribuições

O objetivo desse trabalho é propor e implementar uma infraestrutura para gerenciamento dinâmico de modelos de contexto. Essa infraestrutura deve permitir atualizações em tempo de execução de componentes de modelo.

A principal contribuição desse trabalho é a infraestrutura proposta, a qual representa um estudo de caso de gerenciamento dinâmico de modelos de contexto baseado nos fundamentos da tecnologia de processamento de eventos complexos (CEP). Além disso, a proposta de modularização dos modelos de contexto através da tecnologia OSGi [1] também é uma contribuição desse trabalho, que permite a construção e manutenção dinâmica de uma hierarquia de modelos utilizados por diversas aplicações.

1.2 Metodologia

A metodolgia utilizada para atingir os objetivos desse trabalho é composta pelos seguintes passos:

- Identificação dos elementos que compõem um modelo de contexto, caracterização dos tipos de atualização dinâmica que esses elementos podem sofrer e avaliação

dos impactos que essas atualizações podem provocar em sistemas de gerenciamento dinâmico de modelos contextuais e em aplicações sensíveis ao contexto;

- Levantamento de requisitos para arquiteturas de gerenciamento dinâmico de modelos. Os requisitos devem considerar os impactos mencionados no item anterior;
- Proposta de uma arquitetura que implemente os requisitos mencionados no item anterior;
- Implementação da arquitetura proposta;
- Implementação de um estudo de caso como forma de validar a arquitetura proposta.

1.3 Organização da Dissertação

Essa dissertação é organizada da seguinte maneira.

O Capítulo 2 apresenta um cenário motivacional para gerenciamento dinâmico de modelos contextuais. Além disso, apresenta o modelo de sistema adotado pela infraestrutura de gerenciamento dinâmico de modelos proposta.

O Capítulo 3 apresenta uma introdução ao *Esper* [10], que é uma tecnologia de processamento de eventos complexos na qual a arquitetura para gerenciamento dinâmico de modelos de contexto proposta nesse trabalho é baseada.

O Capítulo 4 discute os requisitos necessários em uma arquitetura de gerenciamento dinâmico de modelos de contexto. Além disso, propõe uma arquitetura que atenda esses requisitos e descreve uma implementação da arquitetura proposta.

O Capítulo 5 apresenta a implementação de um estudo de caso que evidencia a necessidade de atualizações dinâmicas em modelos de contexto. Esse estudo de caso é utilizado como forma de validar a arquitetura proposta através da demonstração de como ela trata os impactos causados por atualizações dinâmicas em componentes de modelo.

O Capítulo 6 discute os trabalhos correlatos ao tema central de pesquisa dessa dissertação.

Por fim, o Capítulo 7 apresenta as conclusões dessa dissertação, as quais evidenciam as principais contribuições e os principais desafios que ficaram em aberto.

CAPÍTULO 2

Gerenciamento Dinâmico de Modelos de Contexto

Esse capítulo faz uma breve discussão sobre modelos de contexto e arquiteturas para sistemas sensíveis ao contexto. Ele discute também um cenário motivacional para gerenciamento dinâmico de modelos contextuais. Além disso, o capítulo caracteriza e limita as atualizações sobre modelos de contexto que são consideradas nesse trabalho, as quais são denominadas atualizações *suaves*. Por fim, os elementos fundamentais que fazem parte de modelos de contexto são identificados, com o objetivo de limitar os componentes de modelos que serão passíveis de gerenciamento dinâmico nas discussões desse trabalho. Essa discussão é concluída com uma análise de possíveis impactos que atualizações sobre modelos podem causar em sistemas sensíveis ao contexto.

2.1 Modelos de Contexto

Modelos de contexto são representações abstratas das situações e condições que são relevantes para o universo de discurso de uma aplicação [9]. Considere, por exemplo, a aplicação de *chat* sensível ao contexto *ConChat*, descrita em [21]. Essa aplicação permite a inferência automática das atividades desenvolvidas por usuários a partir de informações contextuais. Dessa forma, antes de iniciar uma interação via *chat*, ao invés de consultar o *status* de um contato (normalmente, os valores permitidos são *disponível* ou *ocupado*), os usuários podem consultar que atividade esse contato está realizando, o que pode auxiliá-lo a evitar interrupções indesejadas.

Para o universo de discurso da aplicação descrita, as situações relevantes são as atividades desenvolvidas por usuários. Essas atividades são descritas através de um conjunto de condições que restringem valores de informações contextuais associadas aos usuários em questão. A representação dessas atividades e das condições associadas às mesmas são feitas através do modelo de contexto proposto pelos autores, o qual é baseado em lógica de primeira ordem com predicados de quatro argumentos que assumem a

seguinte forma:

$$Context(< ContextType >, < Subject >, < Relater >, < Object >)$$

ContextType é relativo ao tipo de informação contextual que o predicado descreve, *Subject* é a pessoa, lugar ou objeto ao qual *ContextType* é associado, *Object* é um valor relacionado ao *Subject*, e *Relater* é um operador, verbo ou preposição que relaciona *Subject* com *Object*. O exemplo abaixo mostra trechos do modelo de contexto utilizado pelo *ConChat*.

- $Context(\#people, Room2401, >=, 3) \quad \wedge \quad Context(Application, PowerPoint, Is, Running) \quad => Context(RoomActivity, 2401, Is, Presentation)$
- $Context(\#people, Room2401, >=, 3) \wedge NOTContext(Application, PowerPoint, Is, Running) => Context(RoomActivity, 2401, Is, Meeting)$
- $Context(\#people, Room2401, =, 1) \quad \wedge \quad Context(Application, VisualStudio, Is, Running) \quad => Context(RoomActivity, 2401, Is, IndividualDevelopment)$

Nesse exemplo, é possível perceber que as atividades que estão ocorrendo em uma sala podem ser inferidas a partir do número de pessoas que estão ocupando a mesma e do tipo de aplicação executando no dispositivo de um usuário. Dessa forma, para inferir a atividade do usuário em questão, basta criar uma nova construção no modelo que verifique se o mesmo está presente na sala onde a atividade mencionada foi detectada, como por exemplo, $Context(Location, User, Inside, Room2401)$.

O modelo de contexto utilizado para descrever situações de interesse no *ConChat* utiliza elementos de lógica como seu principal fundamento. Porém, existem diversas abordagens de modelagem de contexto que diferem entre si por aspectos como nível de expressividade, formalidade e aplicabilidade para diferentes cenários e tipos de aplicações sensíveis ao contexto. Alguns exemplos dessas abordagens são modelos atributo-valor, orientados a objetos, gráficos, baseados em linguagem de marcação e ontologias. Descrições detalhadas dessas abordagens podem ser encontradas em [2].

A Seção 2.5 apresenta o conceito de *Componentes de Modelo*, que são elementos fundamentais a partir dos quais um modelo é construído. Esse conceito procura definir elementos de um modelo de contexto que estão presentes na maioria das abordagens de modelagem, mas que são abstraídos de forma diferente de acordo com a abordagem adotada.

2.2 Arquiteturas de Sistemas Sensíveis ao Contexto

Com o objetivo de dar suporte ao desenvolvimento de aplicações sensíveis ao contexto, é comum o uso de uma infraestrutura para obtenção, gerenciamento e disseminação de informações contextuais. Henricksen & Indulska[17], por exemplo, propõem um modelo de camadas de uma infraestrutura de software que desempenha essas funcionalidades. A Figura 2.1 ilustra o modelo em questão.

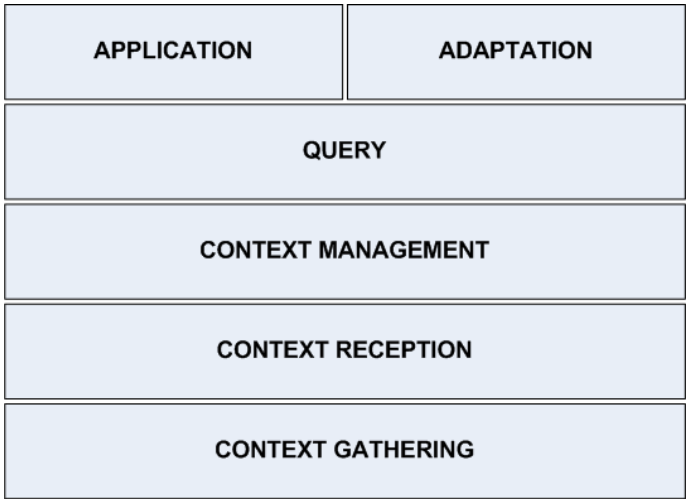


Figura 2.1: *Arquitetura em camadas do sistema sensível ao contexto proposto por Henricksen & Indulska.*

Esse modelo propõe as camadas *Context Gathering* e *Context Reception*, que desempenham a funcionalidade de obtenção de informações contextuais, as quais são produzidas por sensores. A camada *Context Management* é responsável pelo gerenciamento de informações contextuais. Ela mantém um conjunto de modelos de contexto, os quais definem representações para as informações recebidas das camadas inferiores. Essas representações são utilizadas pelos mecanismos de inferência de informações contextuais derivadas, as quais são produzidas de acordo com uma linguagem para descrição de situações de interesse. Por fim, as camadas *Query* e *Adaptation* desempenham a funcionalidade de disseminação de eventos contextuais para as aplicações.

Arquiteturas para computação sensível ao contexto normalmente fundamentam o gerenciamento de informações contextuais em sistemas baseados em eventos, tais como sistemas *publish/subscribe*. Algumas soluções preferem desenvolver o seu próprio sistema baseado em eventos para, dentre outros motivos, evitar limitar a descrição das situações contextuais à linguagem de assinaturas de eventos suportadas por sistemas de propósito geral. Em ambos os casos, a abordagem de modelagem de contexto adotada por uma arquitetura e as consequentes limitações na manutenção dos modelos em tempo de execução são fortemente ligadas à escolha do sistema baseado em eventos na qual a arquitetura é fundamentada.

Essa dissertação trata o problema de manutenção dinâmica de modelos de contexto na perspectiva de tais sistemas, sobretudo com relação ao funcionamento das suas máquinas de inferência de eventos contextuais. Para tal, esse trabalho adota o modelo de sistema ilustrado na Figura 2.2, o qual é baseado no modelo de funcionamento dos *Sistemas de Processamento de Fluxos de Informações*, ou IFPS. Esse termo é utilizado pelos autores de [19] para caracterizar sistemas distribuídos que possuem como requisito o processamento de fluxos de informações, oriundos de fontes geograficamente distribuídas, que devem produzir resultados derivados de regras que avaliam continuamente esses fluxos. Sistemas baseados em eventos são um exemplo de IFPS.

Os IFPS integram geradores (*Sources*) e consumidores (*Sinks*) de fluxos através das máquinas IFP (representadas na figura por *IFP Engine*). Essas máquinas são elementos infraestruturais do modelo de sistema que operam de acordo com um conjunto de regras de processamento, as quais descrevem como os fluxos de entrada devem ser processados para que novas informações sejam produzidas.

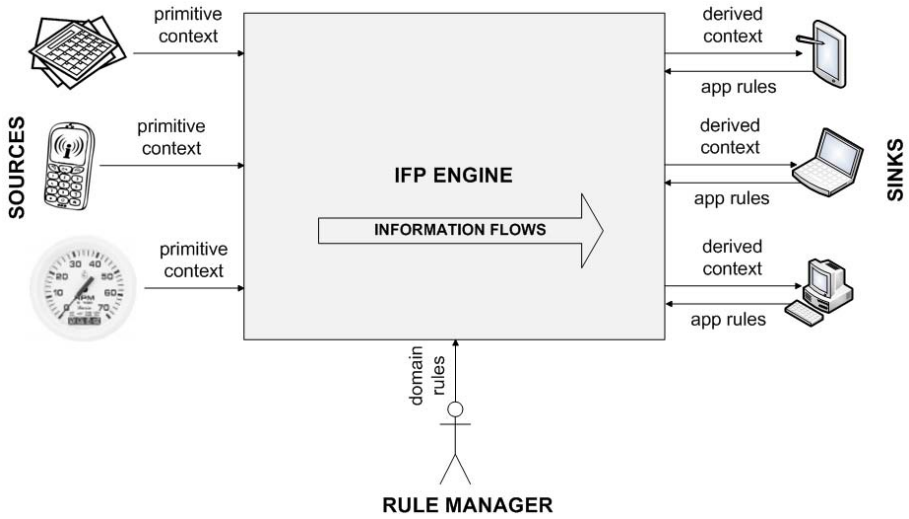


Figura 2.2: *Modelo de funcionamento dos IFPS adaptado para o universo da computação sensível ao contexto.*

Em um IFPS, os *Sources* são responsáveis por criar os fluxos de informações que entram em uma máquina IFP. No modelo de sistemas adotado nesse trabalho, eles atuam como provedores de informações contextuais primitivas, que são informações obtidas diretamente, sem a necessidade de processos de inferência intermediários. Os *Sinks* são os elementos que consomem os fluxos de informação produzidos por uma máquina IFP, e no modelo de sistemas eles representam as aplicações sensíveis ao contexto que registram interesse em eventos contextuais. Em um comparativo com o *ConChat*, os sensores utilizados para inferir as atividades de usuários desempenham o papel de *Sources*, enquanto os clientes de *chat* desempenham o papel de *Sinks*.

A máquina IFP é responsável por processar os fluxos de informações gerados

pelos *Sources*, de acordo com um conjunto de regras, com o objetivo de produzir novos fluxos. No modelo de sistema, esses fluxos são informações contextuais derivadas, ou seja, informações inferidas pelas regras de processamento. As regras podem ser classificadas em *Regras de Domínio* e *Regras de Aplicação*.

As *Regras de Domínio* são criadas por *Rule Managers* e permitem correlacionamento de informações contextuais, produzindo como resultado eventos contextuais derivados. As *Regras de Aplicação* são utilizadas por *Sinks* para descrever o interesse dos mesmos nas informações contextuais derivadas produzidas pelas *Regras de Domínio*. Os *Rule Managers* são pessoas especialistas no domínio de operação de um IFPS, responsáveis por modelar as regras, tipos de eventos contextuais e entidades do cenário sensível ao contexto associados ao domínio do qual são especialistas.

Quando aplicado em computação sensível ao contexto, um IFPS delimita um modelo contextual como sendo as descrições das regras, entidades e dos tipos de eventos mantidos e manipulados por uma máquina IFP. Esses elementos representam a aplicação dos modelos de dados e de regras de um IFPS, segundo a classificação de [19], em um domínio associado a um cenário sensível ao contexto.

2.3 Cenário de Manutenção Dinâmica de Modelos de Contexto

A literatura em computação sensível ao contexto usualmente explora casos em que a modelagem de contexto é estabelecida estaticamente. Nesses casos, as aplicações são desenvolvidas com base em um conhecimento prévio do modelo de contexto, que não pode ser alterado ao longo da execução das aplicações, sob risco de sofrer interrupções ou inconsistências.

Uma mesma informação contextual pode ser inferida de maneiras distintas, dependendo do próprio contexto do usuário, da execução da aplicação ou do ambiente provedor da informação contextual (sensores, agentes de inferência e máquinas baseadas em eventos).

Considere, por exemplo, o contexto “ocupado”, que define a disponibilidade de um usuário, utilizado por uma aplicação de *chat* similar ao *ConChat* para, tanto atualizar um indicador visual para o usuário sobre o estado de cada um dos seus contatos, como para permitir o início de uma comunicação, evitando, assim, interrupções indesejadas. A aplicação é notificada sempre que o estado de um usuário transitar entre “ocupado” e “disponível”. Para inferir esses estados, o modelo de contexto apresentado na Seção 2.1 poderia ser utilizado.

Porém, em cenários mais dinâmicos, é desejável que o modelo mencionado seja atualizado dinamicamente para que o contexto "ocupado" possa ser inferido de diversas maneiras, como informado explicitamente por uma interação com o usuário, baseando-se na sua localização, nas atividades agendadas para aquele período, no estado do local onde ele se encontra (e.g. *está ocorrendo uma reunião*), no estado de uso de seu telefone, só para citar algumas possibilidades.

No caso da inferência baseada em localização, pode-se considerar, por exemplo, que se um usuário se encontra em seu escritório, então a semântica desse local é *local de trabalho*, e o sistema pode utilizar as tarefas programadas na agenda para definir o estado de disponibilidade. Por outro lado, se o usuário em questão está em um hospital, a semântica de *local de trabalho* não deve mais ser aplicada, pois ele assume um papel diferente e próprio daquele local. Deste modo, estar na "sala de espera" pode significar estar livre para o usuário, mas "ocupado" para outro que trabalha naquele local.

Portanto, além do papel do usuário, as regras semânticas do local em que o mesmo se encontra (hospital) são necessárias para determinar qual inferência será aplicada naquele cenário. Enquanto que o primeiro aspecto pode ser pré-definido na aplicação do usuário, o segundo estabelece um dinamismo que depende da localização do mesmo e não é controlado pela aplicação e nem pelo usuário.

Uma arquitetura para computação sensível ao contexto deve permitir que uma mesma aplicação possa, em contextos diferentes, perceber o estado "ocupado" de uma pessoa de acordo com os diferentes significados que esse estado possa assumir. Embora tais semânticas ainda precisem ser modeladas, a escolha e a avaliação da regra de inferência que deve ser aplicada em cada caso ainda é uma tarefa definida estaticamente nas arquiteturas atuais. Esta natureza dinâmica dos modelos sugere a adoção de arquiteturas para sistemas sensíveis ao contexto que suportem modelos dinâmicos sem interferir na consistência do funcionamento das aplicações pré-existentes.

2.4 Atualização Suave em Modelos de Contexto

Atualizações em modelos de contexto, exemplificadas na seção anterior, podem ocorrer por motivos diversos. As principais razões que provocam uma atualização são:

- *evolução natural dos modelos a partir de um modelo de alto nível pré-estabelecido e assumido pelas aplicações* - esse modelo de alto nível é costumeiramente chamado de ontologia de alto nível em alguns trabalhos [26, 25]. Assim como espera-se que os modelos possam evoluir como resultado da inclusão de novos sensores ou a especialização das regras de inferência, as aplicações baseadas nos modelos de alto nível devem manter o funcionamento consistente e adaptado aos modelos,

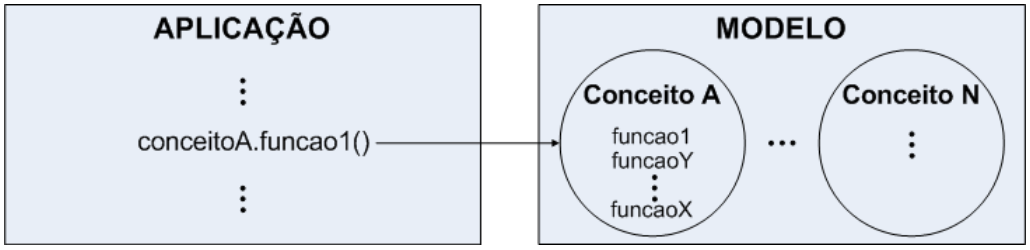
a despeito dessas evoluções. Espera-se que isso não ocorra nas aplicações que são fortemente acopladas aos seus próprios modelos. Neste caso, as aplicações evoluem com os próprios modelos e a interrupção das aplicações é um fenômeno já esperado pelos desenvolvedores;

- *evoluções temporárias* - evoluções temporárias nos modelos ocorrem quando eles precisam ser readequados a um cenário particular, com suas próprias características, mas cuja modificação tem um tempo de vida limitado. Em geral, essas modificações estão associadas à inclusão de novas aplicações, também de uso limitado, e que podem impactar nas situações contextuais utilizadas por aplicações já existentes. O estudo de caso do Capítulo 5 exemplifica essa situação;
- *mobilidade inter-domínio* - em um cenário distribuído, espera-se a segmentação do ambiente em domínios, os quais podem manter suas especializações de modelos e próprias aplicações. A mobilidade dos dispositivos causa às aplicações a necessidade de lidar com modelos diferentes daqueles do seu domínio original. Este problema pode ser tratado como interoperabilidade inter-modelos, como em CoCo [5]. Mas na perspectiva deste trabalho, espera-se que as aplicações sejam baseadas em modelos de alto nível, compartilhados entre os domínios. Essa situação é, do ponto de vista da aplicação, equivalente à do primeiro caso.

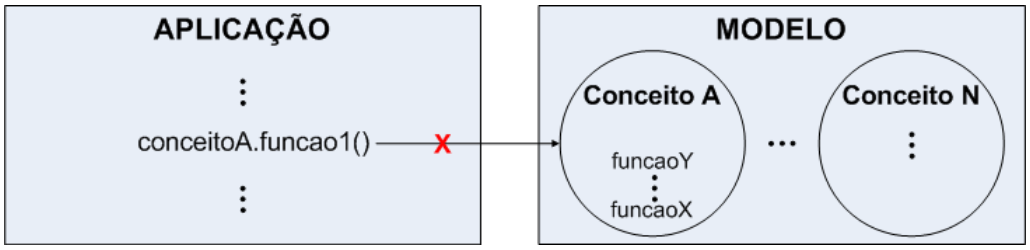
Nos três casos citados, as operações de atualização realizadas sobre modelos podem provocar nas aplicações que os utilizam problemas de inconsistência, causados por incompatibilidades entre novas versões do modelo e versões antigas previamente adotadas por aplicações. Para exemplificar a natureza dessas inconsistências, considere o cenário ilustrado na Figura 2.3.

A Figura 2.3(a) ilustra o uso, por parte de uma aplicação, de uma função (*funcao1*) definida em um conceito (*Conceito A*) descrito no modelo de contexto utilizado por essa aplicação. Na Figura 2.3(b), uma nova versão do modelo é gerada, sendo que a definição da função utilizada pela aplicação foi removida, apesar de uma referência para a mesma ainda existir na aplicação. Dessa forma, se em algum momento essa referência for utilizada pela aplicação, um problema de inconsistência será gerado, pois a definição dessa função já não existe mais.

A origem dos problemas de inconsistência mencionados está no fato de que os conceitos de modelos de contexto utilizados por uma aplicação são definidos externamente às mesmas. Dessa forma, quando uma aplicação é compilada, referências (normalmente denominadas símbolos) para essas definições externas são produzidas. Se durante a execução da aplicação essas referências não forem compatíveis com as respectivas definições concretas, então um problema de inconsistência é gerado. Nesse trabalho, definimos esse problema como *inconsistência simbólica*, e as implicações do mesmo são explicitadas na definição a seguir.



(a) *Cenário inicial de uso dos conceitos de um modelo por uma aplicação.*



(b) *Atualização realizada sobre um conceito do modelo utilizado pela aplicação.*

Figura 2.3: *Inconsistência causada por uma atualização em um modelo de contexto utilizado por uma aplicação.*

Inconsistência simbólica uma aplicação sensível ao contexto precisa ser redesenvolvida ou reimplantada quando uma atualização em um modelo de contexto comprometer a consistência da representação simbólica definida na aplicação em questão.

É importante ressaltar que nem toda atualização em modelos de contexto gera inconsistência simbólica. Por exemplo, no cenário descrito na Figura 2.3, uma nova função poderia ser adicionada no *Conceito A* sem gerar problemas para aplicações que o utilizam. Nesse caso, essas aplicações não precisariam ser redesenvolvidas ou reimplantadas. Nesse trabalho, definimos atualizações dessa natureza como *atualizações suaves*.

Atualização suave atualização em um modelo de contexto que não causa nenhuma inconsistência simbólica nas aplicações em execução que utilizam o modelo em questão.

As situações em que inconsistências simbólicas ocorrem e a classificação das atualizações como sendo suaves ou não são aspectos que dependem das linguagens utilizadas na implementação das aplicações sensíveis ao contexto. Além disso, esses aspectos também dependem das características das máquinas IFP utilizadas para inferência de eventos contextuais e das linguagens de processamento oferecidas por elas no que diz respeito ao processo de resolução de referências simbólicas.

Nas aplicações sensíveis ao contexto, esses dois aspectos são influenciados pelo paradigma da linguagem utilizada no desenvolvimento (fortemente ou fracamente tipado) e pelo modelo de linkedição (dinâmico ou estático) utilizado pela linguagem.

Nas linguagens de processamento de fluxos, esses aspectos são influenciados pela forma com que os símbolos que representam componentes de modelo, utilizados em uma regra, são ligados com as implementações concretas representadas por esses símbolos. Por fim, nas máquinas IFP esses aspectos são influenciados pelo paradigma e pelo modelo de linkedição da linguagem de desenvolvimento da máquina e também pelo modelo de linkedição da linguagem de processamento adotada.

Nesse trabalho, para implementar o sistema de gerenciamento dinâmico de modelos de contexto proposto no Capítulo 4, foi adotada a linguagem Java e a máquina IFP Esper. Dessa forma, os problemas de inconsistência simbólica que podem ocorrer nesse sistema e nas aplicações baseadas nele são relacionados com as características da linguagem Java, que também é utilizada na implementação do Esper, e da linguagem EPL, utilizada pelo Esper para especificar regras de processamento.

2.5 Componentes de Modelo de Contexto

Os autores de [19] utilizam três aspectos para classificar um IFPS, sendo eles arquitetura, modelo de dados e modelo de regras. O modelo de dados define a representação dos elementos que compõem os fluxos que são produzidos por *Sources*, processados pela máquina IFP e consumidos por *Sinks*. O modelo de regras define as características e operadores da linguagem de processamento utilizada para especificar as regras de inferência.

Dessa forma, se aplicados a computação sensível ao contexto, os modelos de dados e de regras podem ser vistos como componentes de modelos de contexto. Nesse trabalho, definimos *Componentes de Modelo de Contexto* como sendo as abstrações fundamentais a partir das quais um modelo é construído. Cada abordagem de modelagem de contexto especifica seus próprios componentes de modelo. Por exemplo, modelagem de contexto baseada em ontologias define *classes* (conceitos) e *relações entre classes* como alguns dos componentes de modelos.

Nesse trabalho, classificamos os componentes de modelo em *eventos contextuais* e *entidades*, que são os elementos que definem a representação de elementos que compõem fluxos de informações contextuais (similar ao modelo de dados de [19]), e *regras* e *operadores*, que são os elementos que definem o modelo de regras e o conjunto de operadores utilizados por uma máquina IFP (similar ao modelo de regras de [19]).

- *Eventos Contextuais* - representam alterações em situações contextuais de interesse de uma ou mais aplicações ou serviços. Por exemplo, no *ConChat*, enquanto "*ocupado*" pode ser modelado como o estado de uma pessoa, uma aplicação tipicamente se interessa com a mudança neste contexto para promover uma adaptação. Este evento pode ser abstratamente descrito como *"avise-me quando o estado do usuário X mudar para "ocupado"*.

- *Entidades* - representam os objetos que compõem o cenário de computação sensível ao contexto e cujo estado é compreendido como informação contextual. No *ConChat*, *Pessoa* é um exemplo de entidade. Entidades podem ser modeladas especificamente para um cenário ou aplicação.
- *Regras* - representam condições, expressões e consultas, descritas segundo uma linguagem definida pela máquina de processamento de fluxos de informações, e que permitem tanto a aplicações como ao restante do sistema indicar situações que produzem novos eventos contextuais associados às respectivas entidades. Tipicamente, regras são específicas de domínios ou de aplicações.
- *Operadores* - representam primitivas de manipulação de eventos contextuais. A semântica de operadores pode ser variável ou fixa. Os operadores de semântica variável normalmente atuam sobre um tipo específico de informação de contexto. Por exemplo, um contexto de localização geográfica pode oferecer o operador *distância*. Os operadores de semântica fixa não dependem do contexto de utilização dos mesmos e, em alguns casos, são pré-definidos pela linguagem da máquina IFP, como, por exemplo, os operadores lógicos. Operadores existem independentemente de aplicações e são utilizados na modelagem de regras.

As diferentes abordagens de modelagem de contexto permitem a especificação parcial ou total desses elementos. O Capítulo 3 apresenta uma discussão sobre como os componentes de modelo definidos nessa seção são modelados em uma máquina IFP baseada na tecnologia de processamento de eventos complexos.

2.6 Impactos do Dinamismo em Componentes de Modelo

Atualizações em modelos de contexto em tempo de execução podem causar inconsistências tanto no funcionamento das aplicações sensíveis ao contexto quanto no processamento de regras pelas máquinas IFP. Tais inconsistências são mais relevantes à medida em que são utilizadas abordagens de modelagem de contexto mais complexas e com rica expressividade. Uma discussão sobre os principais impactos causados por atualizações em componentes de modelos de contexto é feita a seguir.

1. *Reconhecimento de tipos* - Algumas abordagens de modelagem de contexto permitem a criação de novos eventos e entidades como especializações de outras construções pré-existentes. Nestes casos, aplicações devem reconhecer[1] esses novos componentes de modelo à medida em que eles se enquadrarem nas construções de registro de interesse já definidos pela aplicação. Por exemplo, uma aplicação interessada

[1]Esse reconhecimento de novos tipos de construções por parte de uma aplicação normalmente é chamado de *Injeção de Tipos*.

em eventos de uma entidade *Pessoa* também deve receber eventos relativos a uma entidade *Paciente*, se esta for uma especialização daquela. Neste caso, a existência de novas especializações pode estar ligada tanto à atualização dinâmica de um modelo, devido à inclusão de novos sensores, por exemplo, como à interação com outros sistemas (mobilidade entre domínios). Se a abordagem de modelagem de contexto adotada é simples, como em modelos atributo-valor, o impacto da atualização é negligenciável. Entretanto, à medida em que são adotadas abordagens mais complexas e tipadas, tais como orientadas a objetos ou baseadas em ontologias, os efeitos podem ser mais impactantes.

2. *Inconsistência semântica entre eventos* - A adição de novas regras pode fazer com que eventos com semânticas inconsistentes sejam gerados simultaneamente. Considere, por exemplo, a existência de uma regra que defina que uma pessoa está *disponível* sempre que ela estiver na sala X. Um tempo depois, uma nova regra é criada, a qual considera que uma pessoa está *ocupada* sempre que for detectada uma atividade de reunião na sala em que essa pessoa se encontra. Nesse caso, se uma pessoa entrar na sala X e estiver acontecendo uma reunião, os eventos *disponível* e *ocupado* serão produzidos simultaneamente, o que caracteriza uma inconsistência semântica entre eventos.
3. *Problemas no mecanismo de matching* - A máquina de processamento de eventos é responsável por realizar o casamento (*matching*) dos eventos contextuais detectados em um cenário com os interesses de clientes. O modelo de casamento dos eventos adotado por uma máquina está fortemente ligado à linguagem de especificação de interesses ou assinaturas adotada por ela. Em linguagens que não lidam com estado mantido entre eventos, o casamento dos eventos sofre pouca influência da atualização dinâmica dos modelos, pois o casamento é realizado individualmente sobre cada evento disparado. Neste caso, o efeito de alterações dinâmicas nas regras pode ser delegado isoladamente para as implementações das regras. Em linguagens que proveem o conceito de estado entre eventos (*stateful*[2]), uma atualização em modelos pode influenciar o mecanismo de *matching* da máquina de processamento de eventos.
4. *Problemas no mecanismo de filtering* - Máquinas que proveem o conceito de estado entre eventos definem janelas em tempo de execução que restringem os eventos que podem ser envolvidos em um casamento com interesses de clientes. Essa tarefa é chamada de *filtering* na teoria de sistemas baseados em eventos. A função da janela, nesse caso, é definir, dentre os eventos recebidos pela máquina, quais são candidatos a avaliação das expressões dos clientes. Normalmente, isso é feito através da

[2]Ou seja, o mecanismo de *matching* pode considerar o estado de eventos ocorridos anteriormente.

implementação de uma política de seleção, que descarta ou seleciona eventos de acordo com critérios estabelecidos. Atualizações em uma política podem fazer com que o conjunto de eventos que compõem uma janela passe a ser inconsistente com a nova política, pois eventos que foram previamente selecionados para a janela podem não mais estar de acordo com os critérios da política de seleção atualizada. Em máquinas sem estado, o *filtering* precisa ser realizado no lado da aplicação.

A Tabela 2.1 mostra os possíveis impactos causados por atualizações em componentes de modelo juntamente com os agentes impactados pelas atualizações e exemplos de motivos que podem levar a uma atualização.

Componentes	**Possível Impacto**	**Afeta**	**Exemplo de Origem da Atualização**
Eventos Contextuais	***1 e 2***	Sinks	Adição de novos sensores
Entidades	***1***	Sinks	Migração entre domínios
Regras	***2 e 3***	Sinks, Máquina IFP	Criação de novas regras e necessidade da adaptação de regras já existentes para, por exemplo, permitir a implementação do mecanismo de *preferences* proposto por Henricksen & Indulska[17]
Operadores	***3 e 4***	Máquina IFP	Necessidade de alteração da implementação de um operador, como mostrado na Seção 5.3

Tabela 2.1: *Relação entre atualização em componentes de modelo, impactos causados e agentes afetados.*

2.7 Sumário

Esse capítulo apresentou um cenário motivacional para gerenciamento dinâmico de modelos contextuais, o qual explicitou a necessidade de uma arquitetura para computação sensível ao contexto implementar mecanismos que permitam que modelos contextuais sejam atualizados em tempo de execução. Além disso, foi discutido um modelo de sistema, baseado nos sistemas de processamento de fluxos de informações, o qual é utilizado como base para descrição da infraestrutura proposta no Capítulo 4.

Esse capítulo apresentou também os conceitos de inconsistência simbólica e atualização suave. Esses conceitos são utilizados para caracterizar e limitar os tipos de atualização sobre modelos de contexto que são tratados nessa dissertação.

Por fim, o capítulo apresentou uma proposta de classificação dos elementos que compõem um modelo de contexto, chamados nesse trabalho de *componentes de modelo*, e discute os impactos que atualizações sobre esses componentes podem causar tanto nos

sistemas que os gerenciam quanto nas aplicações sensíveis ao contexto que os utilizam. O estudo de caso apresentado no Capítulo 5 demonstra como a abordagem proposta nessa dissertação lida com esses impactos.

No próximo capítulo é feita uma introdução ao Esper, que é uma tecnologia de processamento de eventos complexos na qual a infraestrutura para gerenciamento dinâmico de modelos de contexto proposta nesse trabalho é baseada.

CAPÍTULO 3

Processamento de Fluxos de Informação Utilizando a Tecnologia CEP/Esper

A infraestrutura para gerenciamento dinâmico de informações contextuais proposta nessa dissertação e discutida no próximo capítulo é baseada na máquina IFP Esper, que é uma implementação *open source* do conceito de CEP (*Complex Event Processing*).

CEP é a análise de eventos em tempo real com o objetivo de permitir respostas imediatas a mudanças contextuais [18]. CEP pode ser usado, por exemplo, para definir relacionamentos entre eventos, para detectar padrões complexos entre os mesmos e para correlacionar eventos de diversos fluxos, oriundos, possivelmente, de diversas fontes.

A escolha de CEP, e em particular de Esper, foi motivada pelo fato de essa tecnologia oferecer um dos mais ricos conjuntos de construções, se comparadas a outras máquinas IFP, conforme conclui a classificação proposta em [19]. Consequentemente, os modelos de contextos construídos a partir dos modelos de dados e regras do Esper tendem a ser mais expressivos, se comparados com outros sistemas do mesmo gênero.

3.1 Esper e EPL

O Esper é uma máquina de processamento de fluxos e correlacionamento de eventos que utiliza a linguagem EPL[1] para expressar condições contextuais complexas.

A linguagem EPL é baseada em SQL. Em um paralelo com bancos de dados relacionais, os fluxos de eventos substituem as tabelas como fontes de dados, enquanto os eventos substituem as linhas como unidades básicas de dados.

Para exemplificar o funcionamento da linguagem EPL, considere que um sensor envie para uma instância de uma máquina de eventos notificações periódicas de leituras de temperatura, expressas em graus Fahrenheit, associadas a um determinado ambiente, e que esse ambiente armazene objetos que contenham restrições quanto ao aumento repentino de temperatura, como por exemplo, em um intervalo de um minuto a média

[1] *Event Processing Language*

de temperaturas observadas exceder um determinado limiar expresso em graus Celsius. A regra abaixo mostra um exemplo de como detectar esse tipo de situação utilizando a linguagem EPL.

```
1 insert into
2         TemperaturaElevada
3 select
4         avg(convertFahrenheit2Celsius(tmpF)) as avgTmpC, sensor.name as name
5 from
6         LeituraTemperatura.win:time(60 sec)
7 having
8         avg(convertFahrenheit2Celsius(tmpF)) > threshold
```

As linhas 1 e 2 indicam que um evento do tipo `TemperaturaElevada` deve ser gerado sempre que as restrições especificadas no restante da regra forem satisfeitas. As linhas 3 e 4 indicam os valores que devem ser associados aos atributos `name` e `avgTmpC` das instâncias geradas do tipo de evento `TemperaturaElevada`. Esses valores são derivados do fluxo de eventos gerado pelo sensor de temperatura. Esse fluxo é indicado na regra nas linhas 5 e 6, que também indicam que somente as notificações de temperatura geradas nos últimos sessenta segundos devem ser consideradas para o cálculo da média. Por fim, as linhas 7 e 8 indicam que os eventos do tipo `TemperaturaElevada` só devem ser gerados quando a média de leituras de temperatura, expressas em graus Celsius, exceder um determinado limiar.

3.2 Componentes de Modelo no Esper

Analisando a regra descrita para exemplificar o funcionamento da linguagem EPL, é possível identificar alguns exemplos de componentes de modelo discutidos na Seção 2.5. A tabela a seguir faz o mapeamento de alguns elementos utilizados na regra para os tipos de componentes de modelo correspondentes.

Componente de Modelo	**Construção EPL**	**Referência no Exemplo**
Evento	EventType	TemperaturaElevada
Entidade	propriedades de EventType	sensor
Regra	EPStatement	Toda a consulta EPL
Operação	Single-Row, Aggregated, Views	convertFahreneit2Celsius (semântica variável) e avg (semântica fixa)

Nas próximas seções, uma discussão sobre como esses componentes podem ser modelados no Esper é feita.

3.2.1 Eventos e Entidades

De acordo com o documento de referência do Esper [10], um evento é um registro imutável de uma ocorrência passada de uma ação ou de uma mudança de estado, e que as propriedades de um evento capturam as informações de estado do mesmo. Em

alguns casos, essas propriedades podem ser visualizadas como entidades da classificação proposta na Seção 2.5.

Na regra mostrada anteriormente, por exemplo, o evento *LeituraTemperatura* possui duas propriedades, sendo elas *sensor* e *tmpF*. A propriedade *tmpF* pode ser um tipo primitivo numérico que quantifique o valor de temperatura observado. Já a propriedade *sensor* descreve uma entidade do cenário sensível ao contexto, o que pode exigir uma modelagem mais especializada.

O Esper permite que eventos sejam modelados como classes Java que sigam as convenções JavaBeans [3]. Para isso, o Esper associa cada classe que representa um evento a um tipo de evento, que é uma classe que implementa a interface *EventType*. Esses objetos encapsulam informações a respeito de um certo tipo de evento, como por exemplo nomes de propriedades e seus tipos, super tipos (hierarquia de eventos), classe associadas ao tipo de evento, dentre outras.

O trecho de código a seguir demonstra a criação de um novo tipo de evento no Esper. O objeto *conf* armazena informações de configuração da máquina de eventos. A função *addEventType* cria o novo tipo de evento, identificado por um nome e associado a uma implementação de classe. Por fim, a máquina de eventos é configurada através do objeto *conf* e uma referência para a mesma é armazenada no objeto *engine*.

```
Configuration conf = new Configuration();
conf.addEventType(''LeituraTemperatura'',
        LeituraTemperatura.class);
EPServiceProvider engine = EPServiceProviderManager.
        getDefaultProvider(conf);
```

O Esper permite que novos tipos de eventos sejam criados em tempo de execução. Porém, as classes que serão associadas aos mesmos já devem estar carregadas no *classloader* padrão da máquina virtual em que o Esper está executando antes do início da operação da máquina de eventos. O mesmo vale para entidades, que são componentes dos eventos.

3.2.2 Regras

As regras contextuais são descritas em Esper através da linguagem EPL. Nessa linguagem, as regras podem ser definidas através do registro de consultas na máquina de eventos. Esse registro é realizado através dos objetos do tipo *EPStatement*. O trecho de código a seguir demonstra o registro de uma consulta em Esper.

```
String consulta = insert into TemperaturaElevada ...

EPStatement statement = engine.
        getEPAdministrator().createEPL(consulta);
```

A expressão *insert into* faz com que a máquina de eventos dispare um evento (*TemperaturaElevada*) sempre que uma expressão for avaliada como verdadeira

($avg(convertFahrenheit2Celsius(tmpF)) > threshold$) a partir da análise de um determinado fluxo de eventos (*LeituraTemperatura*). O trecho de código a seguir mostra os procedimentos necessários para gerar notificações e, possivelmente, tomar ações, sempre que um evento do tipo *TemperaturaElevada* for detectado.

```
String consultaTmpEl =
``select * from TemperaturaElevada'';

EPStatement staTmpEl = engine.getEPAdministrator().
        createEPL(consultaTmpEl);

staTmpEl.addListener(new TemperaturaListener());
```

O método *addListener* de um objeto do tipo *EPStatement* recebe como parâmetro um objeto que implemente a interface *UpdateListener*. Essa interface possui o método *update*, que será invocado sempre que a condição contextual descrita na consulta for satisfeita. É nesse método que as ações desejadas para a ocorrência de determinado evento são implementadas.

O Esper não disponibiliza mecanismos para atualizar uma consulta em tempo de execução. Em outras palavras, não é possível alterar a estrutura do objeto *EPStatement* gerado pelo registro de uma consulta.

3.2.3 Operadores

Somente operadores de semântica variável são passíveis de gerenciamento dinâmico, pois operadores de semântica fixa possuem uma funcionalidade muito bem definida e normalmente não representam pontos de extensão de uma máquina IFP. Por isso, essa seção se restringe a esses tipos de operadores oferecidos pelo Esper, sendo eles: *Single-Row Functions*, *Aggregation Functions* e *Views*.

Single-Row Functions são funções que atuam sobre atributos de uma única instância de evento. São definidas através de métodos públicos e estáticos de uma classe Java. O operador *convertFahrenheit2Celsius*, utilizado na regra do trecho de código mostrado a seguir, é um exemplo de *Single-Row Functions*. Considerando a existência da classe Java *br.ufg.inf.Conversor* que implemente o método público e estático *convertFahrenheit2Celsius*, então essa função pode ser registrada na máquina de eventos de acordo com o trecho de código da descrito a seguir.

```
Configuration conf = new Configuration();

conf.addPlugInSingleRowFunction(
``convertFahrenheit2Celsius'', ``br.ufg.inf.Conversor'', ``convertFahrenheit2Celsius'');
```

Aggregation Functions são funções que atuam sobre atributos de um conjunto de eventos. São definidas através da extensão da classe *AggregationSupport*. O operador

avg utilizado na primeira consulta descrita nessa seção é um exemplo de *Aggregation Function* que calcula médias matemáticas.

Por fim, as *Views* são utilizadas para derivar informações de um fluxo de eventos ou para representar janelas de dados sobre os mesmos. São definidas através das extensões das classes *ViewFactorySupport* e *ViewSupport*. O operador *win:time* utilizado na primeira consulta descrita nessa seção é um exemplo de *View*. Ele representa uma janela de dados que retem os eventos do tipo *TemperaturaElevada* detectados nos últimos sessenta segundos.

Assim como os operadores do tipo *Single-Row Functions*, *Aggregation Functions* e *Views* também podem ser customizados e incorporados à máquina de eventos.

O Esper permite que novos tipos de operadores sejam criados em tempo de execução. Assim como no caso de eventos e entidades, as classes que serão associadas aos mesmos já devem estar carregadas no classloader padrão da máquina virtual em que o Esper está executando antes do início da operação da máquina de eventos. Porém, ele não provê suporte para atualizações dinâmicas de implementações associadas a esses operadores.

3.3 Sumário

Esse capítulo apresentou uma breve introdução ao Esper, que é a máquina IFP na qual a infraestrutura para gerenciamento dinâmico de modelos proposta nesse trabalho é baseada. Além disso, foi mostrado como componentes de modelo podem ser definidos e utilizados no Esper. Por fim, foi mostrado que o Esper possui algumas limitações que fazem com que componentes de modelo não possam ser atualizados em tempo de execução.

O próximo capítulo apresenta detalhes da infraestrutura para gerenciamento dinâmico de modelos de contexto proposta nesse trabalho. Além disso, é apresentado o Esper Dinâmico, que é uma adaptação do Esper, desenvolvida pelo autor dessa dissertação, que elimina algumas das limitações do Esper discutidas nesse capítulo com o intuito de permitir que componentes de modelo sejam atualizados dinamicamente.

CAPÍTULO 4

Infraestrutura para Gerenciamento Dinâmico de Modelos de Contexto

Esse capítulo apresenta requisitos, arquitetura e implementação de um sistema de gerenciamento dinâmico de modelos de contexto (SGDMC). Os requisitos apresentados foram levantados com o objetivo de guiar a definição arquitetural e a correspondente implementação no sentido de evitar os impactos causados pelo dinamismo de componentes de modelo. A arquitetura apresentada é fundamentada no modelo de sistema descrito na Seção 2.2, e a implementação da mesma é baseada nas tecnologias OSGi e Esper.

4.1 Requisitos

Arquiteturas para gerenciamento dinâmico de modelos de contexto devem lidar com desafios que não podem ser negligenciados. Dentre eles, os principais são os problemas gerados por inconsistências simbólicas e os impactos causados pelo dinamismo inerente aos componentes de modelo de contexto. Para lidar com esses desafios, uma infraestrutura para gerenciamento dinâmico de modelos de contexto deve implementar os requisitos especificados a seguir.

Modularização dos modelos de contexto - a arquitetura proposta nessa seção tem como um de seus objetivos fornecer suporte para o desenvolvimento de diversos tipos de aplicações sensíveis ao contexto. Essas aplicações podem possuir diferentes requisitos no que diz respeito ao nível de abstração e de granularidade dos componentes descritos pelos modelos de contexto a serem utilizados. Assim sendo, é necessário que os modelos sejam organizados de forma hierárquica, onde modelos mais próximos do topo da hierarquia normalmente definem componentes de alto nível, enquanto modelos mais próximos da base definem componentes mais especializados (ou de baixo nível), os quais dependem dos componentes de alto nível. Essa forma de organização dos modelos se caracteriza pelas necessidades de desacoplamento entre modelos e aplicações, pelo compartilhamento dos modelos por diversas

aplicações e pela presença de relações de dependência entre modelos. Essas necessidades podem ser sanadas através da modularização do SGDMC, onde os modelos podem ser implementados como módulos responsáveis por fornecer definições de componentes de modelo para outros elementos da arquitetura.

Criação, atualização e remoção dinâmica de modelos de contexto - a arquitetura do SGDMC deve implementar mecanismos que permitam, em tempo de execução, que modelos surjam, evoluam e, no caso de evoluções temporárias, sejam removidos. Dessa forma, modelos devem possuir um ciclo de vida bem definido, onde as transições entre os estados desse ciclo não devem causar inconsistências simbólicas nos componentes que utilizam os modelos e nem problemas de dependência entre modelos.

Propagação distribuída de alterações em modelos - como os modelos de contexto devem ser desacoplados dos componentes que os utilizam, então a arquitetura deve implementar mecanismos que permitam a propagação das alterações realizadas sobre modelos. Esses mecanismos devem operar tanto em caráter local quanto em caráter distribuído, ou seja, mudanças realizadas em modelos devem ser propagadas tanto para componentes que operam no mesmo ambiente de execução do SGDMC quanto para componentes externos, como por exemplo os *Sinks*.

Processamento ininterrupto de regras - alterações causadas por mudanças no ciclo de vida de um modelo não podem interromper o processamento de regras em execução, mesmo que essas alterações sejam realizadas sobre componentes de modelo que estejam sendo utilizados no contexto de execução dessas regras.

4.2 Gerenciamento de Modelos de Contexto

Para o mecanismo de gerenciamento de modelos proposto nesse capítulo, modelos de contexto são conjuntos de componentes de modelo que podem definir dependências entre si. Dependência significa fazer uso dos componentes de um modelo para para definir componentes de outro modelo. Por exemplo, entidades e eventos presentes em um modelo *A* podem ser utilizados para definir entidades e eventos de um modelo *B* através de relações hierárquicas ou de composições que definem componentes de modelo mais complexos através de componentes mais simples. Um outro exemplo é a utilização de um operador presente em um modelo *A* para realizar uma sub tarefa na implementação de um operador definido em um modelo *B*.

A existência de relações de dependência entre modelos faz com que os mesmos sejam organizados de forma hierárquica, conforme ilustrado na Figura 4.1. Nessa figura, as setas tracejadas indicam o uso de um modelo por uma aplicação, enquanto as setas contínuas indicam dependências entre modelos. Um aspecto a se notar é que essa forma

de organização dos modelos proporciona que diferentes tipos de aplicação (por exemplo, APP1, APP2 e APP3 indicadas na figura) possam utilizar diferentes modelos da hierarquia.

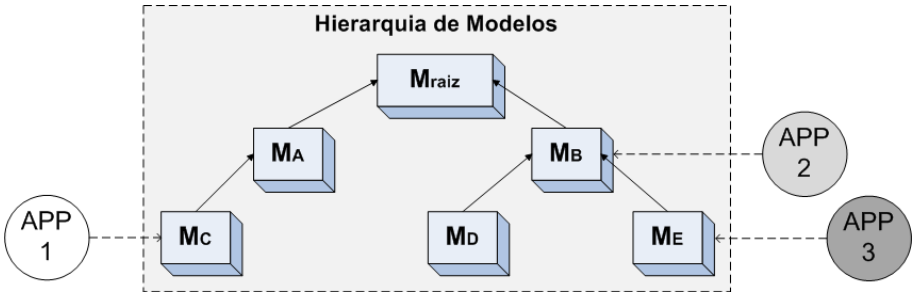


Figura 4.1: *Organização hierárquica dos modelos de contexto.*

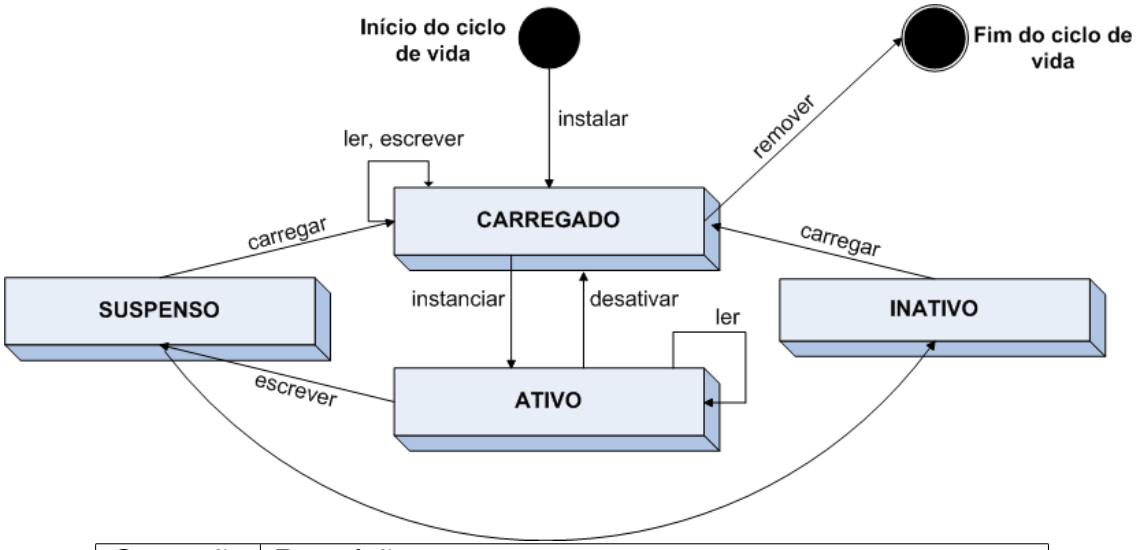
O mecanismo de gerenciamento de modelos proposto deve permitir a criação de novos modelos e a atualização ou remoção de modelos já existentes sem que essas operações gerem inconsistências simbólicas nos componentes arquiteturais que os utilizam e nem inconsistências nas relações de dependência entre modelos. Para explicar o funcionamento do mecanismo mencionado, esse trabalho propõe um ciclo de vida para modelos de contexto, ilustrado na Figura 4.2.

Um modelo no estado Carregado está inserido no ambiente de execução de um SGDMC, o que é feito pela operação *instalar*, e todas as dependências entre modelos relativas ao mesmo foram resolvidas, o que é feito pela operação *carregar*. Resolver as dependências significa verificar se os componentes descritos por modelos dos quais se depende estão disponíveis para uso, o que ocorre quando esses modelos estão instalados e carregados. A operação *remover*, que exclui os componentes descritos em um modelo do ambiente de execução de um SGDMC, só pode ser aplicada a modelos que se encontram no estado Carregado, pois é garantido que os mesmos não estão sendo utilizados.

Um modelo no estado Ativo está sendo utilizado, através da operação *instanciar*, por pelo menos um componente da arquitetura. Caso o sistema detecte que um modelo no estado Ativo não está sendo utilizado, então esse modelo entra no estado Carregado através da operação *desativar*.

Um modelo entra no estado Suspenso quando ele estava sendo utilizado por algum componente da arquitetura e alguma operação do tipo *escrever* foi executada. Esse é um estado implícito, pois modelos não param nesse estado, e a mudança para outros estados deve ser acompanhada de verificações de inconsistências nas relações de dependência entre modelos. Caso esses problemas não ocorram, a operação *carregar* será invocada e o modelo passará para o estado Carregado. Caso contrário, o modelo irá para o estado Inativo.

Um modelo no estado Inativo sofreu uma operação de escrita, foi suspenso e não pôde ser recarregado. Essa situação indica que um modelo sofreu uma alteração que



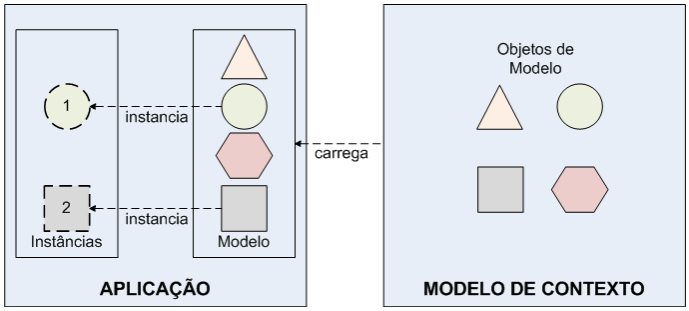
Operação	Descrição
Carregar	Resolve dependências entre modelos
Instalar	Insere os componentes de um modelo no ambiente de execução de um SGDMC e invoca a operação *carregar*
Escrever	Adiciona, altera ou remove componentes associados a um modelo de contexto
Remover	Remove um modelo do ambiente de execução de um SGDMC
Instanciar	Cria instâncias de componentes de um modelo
Desativar	Detecta que um modelo não está sendo utilizado
Ler	Lê o conteúdo de um modelo de contexto

Figura 4.2: *Ciclo de vida de um modelo de contexto.*

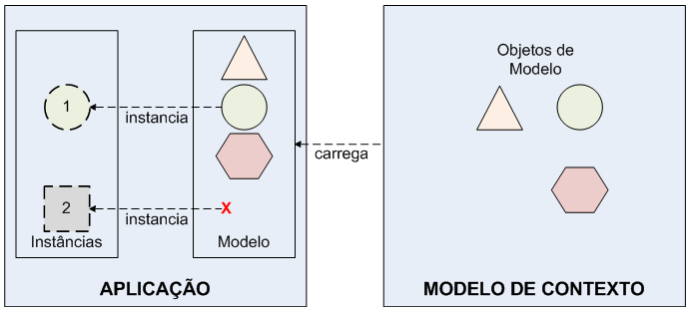
causou algum problema nas relações de dependência do modelo em questão. Nesse caso, o modelo só poderá ser recarregado quando o problema for resolvido.

As operações de escrita podem causar inconsistências em modelos. Essas operações são dividas em adição, alteração e remoção de componentes de modelo. As operações de adição não causam inconsistência simbólica, mas podem causar problemas de dependência entre modelos caso o componente de modelo adicionado dependa de um outro modelo que não está instalado. As operações de alteração e remoção aplicadas a um modelo de contexto podem causar inconsistências simbólicas, exceto para os casos de alterações que se enquadram na categoria de atualização suave. A Figura 4.3 ilustra um cenário onde operações do tipo alteração e remoção provocam uma inconsistência simbólica. A Figura 4.3(a) ilustra o cenário inicial, onde uma aplicação instanciou alguns componentes do modelo utilizado por ela.

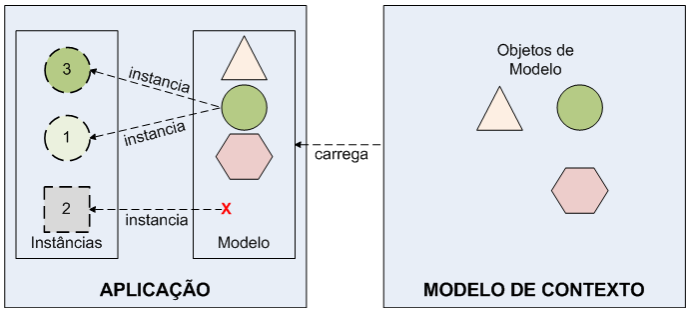
Na Figura 4.3(b), quando o componente de modelo quadrado é removido, verifica-se que a *instância 2* perde a referência para a definição do componente de modelo correspondente, o que pode causar uma inconsistência quando essa instância for referenciada ao longo da execução da aplicação em questão. Na Figura 4.3(c), quando o com-



(a) *Modelo inicialmente carregado.*



(b) *Operação de remoção executada.*



(c) *Operação de alteração executada.*

Figura 4.3: *Operações de escrita executadas sobre um modelo de contexto utilizado por uma aplicação.*

ponente de modelo círculo é alterado, verifica-se que novas instâncias desse componente (por exemplo a instância 3) poderão ser criadas normalmente. Porém, instâncias antigas e referenciadas pela aplicação passarão a conter uma definição inconsistente, o que fica evidenciado pela tonalidade diferente da instância 2 com relação à sua definição de componente de modelo. Considere, por exemplo, que a atualização gerada no componente círculo da Figura 4.3(c) foi a remoção de um atributo de uma entidade. Logicamente, esse atributo continuará existindo na instância 2. Porém, se um trecho de código da aplicação acessar esse atributo um problema ocorrerá, pois a definição do mesmo foi removida.

4.3 Arquitetura e Implementação

A infraestrutura de software para gerenciamento dinâmico de modelos de contexto proposta nesse trabalho é baseada na tecnologia de processamento de eventos Esper e na plataforma de serviços OSGi. O OSGi é um conjunto de especificações que definem um modelo de desenvolvimento onde aplicações escritas na linguagem de programação Java são dinamicamente compostas por diversos módulos, que são chamados de *bundles*. Um *bundle* pode esconder a sua implementação dos demais *bundles* que compõem um sistema, sendo que a comunicação entre os mesmos ocorre através de invocações a serviços. Um serviço é uma funcionalidade oferecida por um módulo. Em OSGi, serviços são objetos Java normalmente associados a uma ou mais interfaces. Para que um *bundle* invoque um serviço oferecido por outro *bundle*, basta que essas interfaces sejam conhecidas.

O fato da utilização de serviços ser baseada em interfaces está relacionado com a característica dos *bundles* de esconderem as suas implementações uns dos outros. Como utilizar serviços requer apenas o conhecimento das interfaces associadas ao mesmos, então é possível que as implementações desses serviços sejam atualizadas dinamicamente de forma transparente àqueles que os utilizam. Para tanto, o OSGi oferece um mecanismo para gerenciamento dinâmico do ciclo de vida de *bundles*. Esse mecanismo permite que *bundles* sejam instalados, atualizados e removidos sem que, para isso, outros *bundles* tenham que ser parados.

Na arquitetura proposta nesse capítulo, os principais papéis desempenhados pelo OSGi são definir e gerenciar dependências entre modelos e permitir o carregamento dinâmico de classes que representam componentes de modelo na Máquina Virtual Java.

A Figura 4.4 ilustra a arquitetura da infraestrutura de software proposta, a qual é baseada no modelo de sistemas apresentado na Seção 2.2. Essa arquitetura é formada por quatro camadas, sendo que cada uma delas define uma responsabilidade específica a ser cumprida pelos componentes que as integram, que são *bundles* OSGi.

A camada *Gerenciamento de Modelos de Contexto* é responsável por controlar o ciclo de vida dos modelos de contexto e por fornecer definições de componentes de modelo para uso dos outros *bundles* da arquitetura e dos *Sinks*. A camada *Gerenciamento de Eventos de Entrada* é responsável pela comunicação com os *Sources* e pelo envio dos fluxos de eventos de entrada para processamento da máquina de eventos. A camada *Processamento de Eventos* é responsável por produzir fluxos de eventos de saída de acordo com um conjunto de *Regras de Domínio*. Ela é formada pelo Esper Dinâmico, que é uma adaptação do Esper que permite a atualização dinâmica de operadores de semântica variável. O Esper Dinâmico possui uma representação própria dos componentes de modelo definidos na camada *Gerenciamento de Modelos de Contexto* (*Single-Row Functions, Aggregation Functions, Views e Event Types*). Por fim, a camada *Gerenciamento de Eventos*

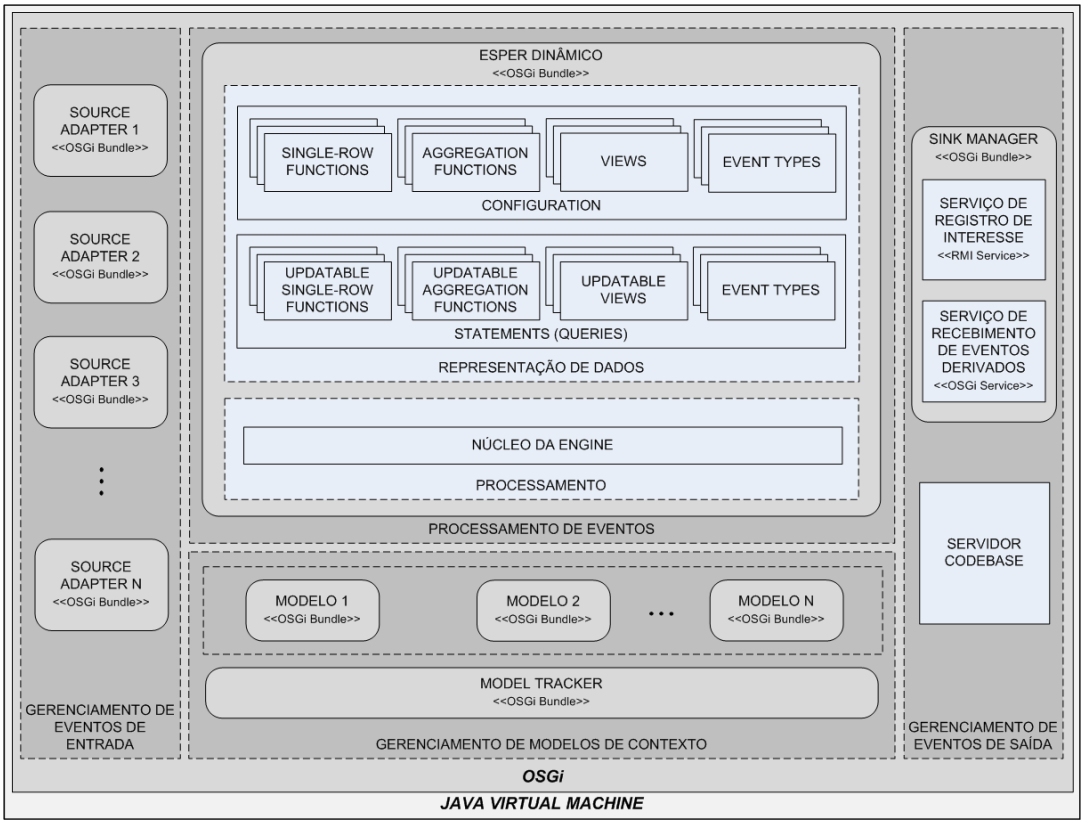


Figura 4.4: *Arquitetura da infraestrutura de gerenciamento dinâmico de modelos proposta.*

de Saída é responsável por registar interesses dos *Sinks* nos eventos contextuais (fluxos de eventos de saída) produzidos pelo Esper Dinâmico e por notificar aos mesmos possíveis ocorrências desses eventos.

As próximas seções discutem em detalhes a implementação de cada uma dessas camadas.

4.3.1 Camada de Gerenciamento de Modelos de Contexto

A realização de um modelo de contexto consiste em transformar a especificação conceitual do mesmo em uma representação dependente de tecnologia. Considere um modelo de contexto orientado a objetos especificado através de diagramas de classes UML. Um exemplo de realização desse modelo é a criação de classes Java derivadas da especificação UML, as quais podem ser utilizadas por aplicações sensíveis ao contexto para representar informações contextuais.

O gerenciamento dinâmico de modelos contextuais proposto neste trabalho divide o processo de realização de modelos em duas etapas. A primeira etapa consiste na instalação dos módulos OSGI que representam os modelos, enquanto a segunda etapa consiste na conversão dos componentes de modelo contidos nesses módulos em representações utilizadas internamente pelo Esper Dinâmico.

Na primeira etapa desse processo ocorre a instalação do *bundle* que representa um modelo no *framework* OSGi, o que é feito por um *rule manager*. Em seguida, o sistema detecta que o modelo em questão foi instalado e, automaticamente, invoca a API de configuração do Esper Dinâmico com o objetivo de converter as definições de componentes de modelo contidas no *bundle* em questão em uma representação utilizada pelo Esper Dinâmico, as quais são ilustradas na Figura 4.4 pelas caixas *single-row functions*, *aggregation functions*, *views* e *event types*. Esse procedimento é realizado pelo componente *Model Tracker*.

O *Model Tracker* tem o seu funcionamento baseado em um padrão muito utilizado no universo OSGI, que é o padrão extensor [16]. A principal ideia por trás desse padrão é modelar extensibilidade dinâmica através dos eventos de ciclo de vida de outros *bundles*. Normalmente, implementações desse padrão consideram a existência de dois tipos de *bundles*, sendo eles o *bundle extensor* e o *bundle de extensão*. O primeiro detecta eventos de ciclo de vida associados ao segundo, atuando como um *listener*. Através da leitura de metadados que descrevem como os recursos do *bundle de extensão* devem ser utilizados, o *bundle extensor* estende uma aplicação.

Bundles OSGi são empacotados como arquivos JAR, e eles definem informações descritivas sobre si[1], ou metadados, no arquivo `MANIFEST`, o qual que deve estar presente em todo JAR. Na implementação da arquitetura proposta, modelos de contexto são *bundles* OSGI que especificam metadados com as seguintes características:

- *Cabeçalhos customizados* - além dos cabeçalhos padronizados pelo OSGi, um bundle que representa um modelo de contexto deve especificar os cabeçalhos customizados descritos na Tabela 4.1, os quais são utilizados pelo componente *Model Tracker* com a finalidade de gerenciamento de operações de atualização sobre modelos mantidos pelo Esper Dinâmico.
- *Exportação de pacotes sincronizada com cabeçalhos customizados* - um *bundle* que representa um modelo deve exportar todos os pacotes que contenham classes especificadas nos cabeçalhos `ContextModel-EventTypes` e `ContextModel-Operators`. Isso deve ocorrer porque essas classes devem ser visíveis para os outros *bundles* que as utilizam. Em especial, essa visibilidade deve existir entre *bundles* que representam modelos, pois é através dela que modelos de baixo nível podem especializar os componentes descritos em modelos de alto nível.

O *Model Tracker* escuta por eventos de ciclo de vida gerados pelo *framework* OSGI. Quando um evento associado a um *bundle* que contenha em seu `MANIFEST`

[1] Alguns exemplos dessas informações são nome do *bundle*, dependências com relação a outros *bundles* e pacotes exportados.

Cabeçalho	Definição
`Bundle-Type: ContextModel`	faz com que o *bundle* seja identificado como um modelo de contexto
`ContextModel-EventTypes`	define uma lista de atributos par/valor (nome do tipo de evento/classe utilizada na definição do evento) que representam os tipos de eventos contidos no modelo
`ContextModel-Rules`	define uma lista de atributos par/valor (nome da regra/especificação EPL) que representam as regras contidas no modelo
`ContextModel-Operators`	define uma lista de atributos que representam os operadores de semântica variável contidos em um modelo

Tabela 4.1: *Cabeçalhos adicionais de um bundle que o caracterizam como um modelo de contexto.*

o cabeçalho `Bundle-Type: ContextModel` é detectado, o *Model Tracker* invoca, de acordo com o tipo de evento detectado, os métodos adequados (por exemplo, adicionar ou remover tipos de eventos, atualizar uma regra) na API de configuração do Esper Dinâmico, sendo que os tipos de evento de interesse são instalação, atualização ou desinstalação de *bundles*. Os parâmetros passados na invocação são lidos dos cabeçalhos `ContextModel-EventTypes`, `ContextModel-Rules` e `ContextModel-Operators`, os quais descrevem, respectivamente, os componentes de modelo *eventos contextuais* e *entidades*, *regras* e *operadores*.

A Listagem 4.1 ilustra um exemplo de arquivo `MANIFEST` que descreve os metadados associados a um modelo. Se o *bundle* correspondente ao modelo em questão fosse instalado em um *framework* OSGi com um SGDMC em execução, o *Model Tracker* detectaria que esse *bundle* corresponde a um modelo e invocaria os métodos `addEventType`, `createEPL` e `addUpdatablePlugInSingleRowFunction` no Esper Dinâmico, utilizando como parâmetro, respectivamente, as descrições contidas nos cabeçalhos `ContextModel-EventTypes`, `ContextModel-Rules` e `ContextModel-Operators`.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Sample Model
Bundle-SymbolicName: inf_samplemodel
Bundle-Version: 1.0.0
Export-Package: inf.samplemodel
                .
                .
                .
ContextModel-EventTypes: Busy;inf.samplemodel.Busy, Available;inf.samplemodel.Available
ContextModel-Rules: r01_Busy;select * from Busy, r02_Available;select * from Available
ContextModel-Operators: singlerow[inside;inside_method;inf.samplemodel.GeoFunctions,
distance;distance_method;samplemodel.GeoFunctions], aggregate[...], view[...]
```

Listagem 4.1: *Exemplo de `MANIFEST` que descreve metadados associados a um modelo*

Estado do ciclo de vida de um modelo	Estado do ciclo de vida de um *bundle*
Carregado	*Resolved*
Ativo	*Active*
Inativo	*Installed*
Suspenso	$Active \rightarrow Stopping \rightarrow Resolved \rightarrow Installed$

Tabela 4.2: *Estados equivalentes dos ciclos de vida de bundles e modelos.*

Nem toda mudança de estado no ciclo de vida de um *bundle* interessa ao componente *Model Tracker*. As mudanças de interesse são aquelas que ocorrem em estados do ciclo de vida que podem ser mapeados para estados do ciclo de vida de modelos. A Figura 4.5 ilustra o ciclo de vida de um *bundle* OSGi, enquanto a Tabela 4.2 mostra as equivalências de estado entre os dois ciclos de vida mencionados.

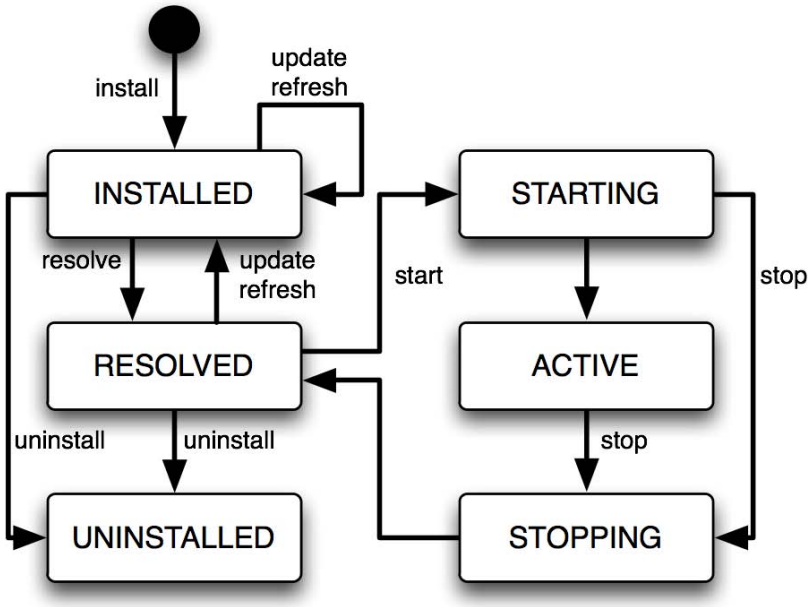


Figura 4.5: *Ciclo de vida de um bundle OSGi.*

Na Tabela 4.2, o estado Suspenso de um modelo não tem um estado equivalente em OSGi, mas sim, um conjunto de transições entre estados. Isso ocorre porque, como já foi dito na Seção 4.2, o estado Suspenso é transitório, e a sua finalidade é indicar que a mudança para um próximo estado dependerá de verificações de inconsistências nas relações de dependência entre modelos, o que é feito pelo próprio *framework* OSGi.

4.3.2 Camada de Processamento de Eventos

A camada de processamento de eventos é formada pelo Esper Dinâmico. O Esper Dinâmico é uma adaptação do Esper que permite que implementações dos operadores

single-row functions, *aggregate functions* e *views* sejam alteradas dinamicamente, de forma que uma atualização possa surtir efeito tanto para *statements* que surgirão após a aplicação da mesma quanto para *statements* já em execução. Como visto na Seção 2.6, atualizações relacionadas com componentes de modelo dos tipos *eventos contextuais* e *entidades* não impactam o funcionamento das máquinas de eventos. Por isso, não foram necessárias alterações no Esper para que o dinamismo associado a esses componentes fosse obtido.

Na Seção 3.2, foi mostrado que o Esper oferece suporte para que novas regras sejam criadas em tempo de execução. Ele permite que isso ocorra também com eventos contextuais, entidades e operadores de semântica variável, que são representados através de classes Java. Porém, o Esper não oferece um mecanismo para que as definições correspondentes a esses componentes de modelo sejam carregadas na Máquina Virtual Java e reconhecidas dentro do ambiente de execução do Esper. Na implementação proposta, isso é obtido através da execução do Esper no *framework* OSGi. Nesse caso, o papel do OSGi é permitir que novas definições de componentes de modelo representados através de classes Java sejam carregadas dinamicamente e reconhecidas pelo ambiente de execução do Esper através das operações de atualização de *bundles*.

A Figura 4.6 ilustra o processo de criação e uso de um operador de semântica variável pelo Esper. No momento da criação de um *statement*, o componente de configuração do Esper é consultado com o objetivo de identificar e configurar os recursos adequados para o *statement* em questão. Se a regra relativa ao mesmo faz uso de operadores de semântica variável, então uma consulta é realizada com o objetivo de identificar as classes que implementam as funcionalidades dos respectivos operadores. Os resultados dessa consulta são armazenados pelo *statement* e utilizados no processamento de eventos. Quando o evento *EVENT_A* é inserido no contexto de execução do *statement STATEMENT_A* para ser processado, em algum momento o operador *operatorA* deverá ser invocado. Nesse momento, o que de fato ocorre é que a função *DoSomething* é invocada em *OperatorAImpl*. Assim sendo, atualizar dinamicamente o comportamento de um operador significa alterar a implementação da função *DoSomething* em tempo de execução.

Para resolver esse problema, foi adotada uma abordagem que utiliza o padrão de projeto *Proxy*, descrito em [13]. Nesse padrão, um *proxy* é um objeto que controla, ou intermedeia, o acesso a um outro objeto. A Figura 4.7 ilustra essa situação, onde o elemento *Client* invoca a função *Operation* no *Proxy*, o qual possui uma referência para *Real Subject*, que é o elemento com o qual *Client* realmente deseja interagir. Quando recebe a invocação, o *Proxy* a delega para *Real Subject*.

A solução implementada no Esper Dinâmico utiliza *proxies* para, no contexto de execução de um *statement*, delegar as chamadas a operadores para os elementos que contém as implementações reais, como mostra a Figura 4.8. Dessa forma, quando

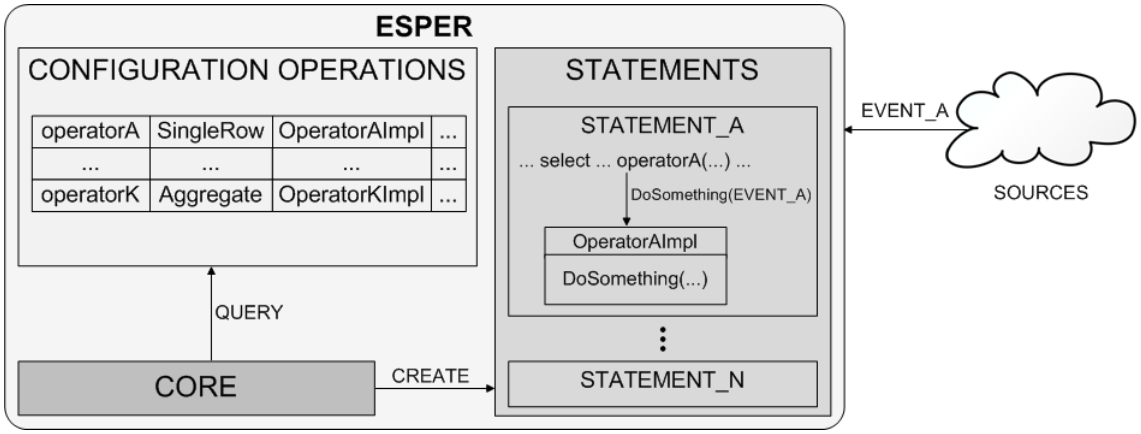


Figura 4.6: *Criação e uso de operadores de semântica variável no Esper.*

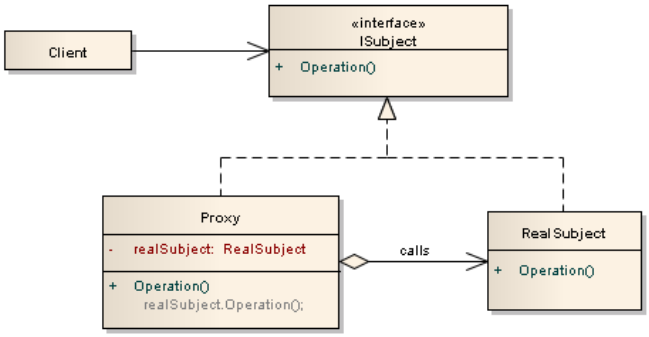


Figura 4.7: *Exemplo de uso de um proxy.*

um desenvolvedor desejar criar um novo tipo de operador de semântica variável no componente de configuração do Esper Dinâmico, um *proxy* deve ser fornecido a esse componente. Porém, os *proxies* são criados de forma transparente ao desenvolvedor como um processo intermediário do mecanismo de criação de novos operadores.

A Listagem 4.2 mostra um fragmento da API de configuração do Esper Dinâmico para criação de novos operadores. Sempre que um dos métodos listados for invocado, um *proxy* será criado e a API do Esper para criação de operadores será invocada em seguida. Quando um desenvolvedor de uma aplicação baseada em Esper deseja criar um novo tipo de operador, além de outras informações, são fornecidos um nome e uma indicação da classe que implementa a funcionalidade do respectivo operador. Nesse caso, o *proxy* será indicado como a classe que implementa a funcionalidade do operador correspondente.

```
public void addUpdatablePlugInSingleRowFunction(String functionName,
                String className, String methodName);
public void addUpdatablePlugInAggregationFunction(String functionName, String className);
public void addUpdatablePlugInView(String namespace, String functionname,
                String viewFactoryClassName, List<String> viewClassesNames);
```

Listagem 4.2: *Trecho da API de configuração do Esper Dinâmico responsável pela criação de novos operadores*

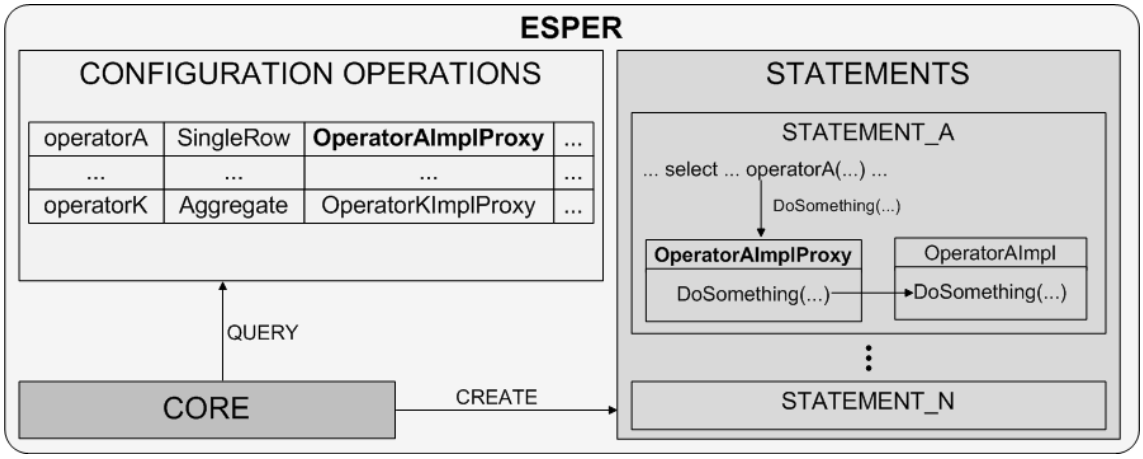


Figura 4.8: *Criação e uso de operadores de semântica variável no Esper com uso de proxy.*

O processo de criação de um *proxy* utiliza manipulação de *bytecodes* e carregamento dinâmico de classes através do uso da ferramenta *javassist* [6]. Manipulação de *bytecodes* é o processo de alterar o conteúdo de arquivos binários denominados *class files*, que são os arquivos utilizados por Máquinas Virtuais Java para executar o carregamento de classes. O *javassist* permite que diversos aspectos de um *class file* sejam redefinidos, como por exemplo, o nome correspondente a uma classe. Além disso, permite a adição, alteração e remoção de métodos, construtores e campos, dentre outras funcionalidades. Os resultados da manipulação de um *class file* podem ser carregados dinamicamente pela máquina virtual como uma nova classe, desde que seja respeitada a restrição de que duas classes com o mesmo nome completo não sejam carregadas pelo mesmo *class loader*.

Sempre que um dos métodos da Listagem 4.2 é invocado, o *class file* da classe que implementa a funcionalidade do operador em questão é utilizado como base para a construção de uma nova classe, a qual desempenhará o papel de *proxy*. Essa nova classe é uma cópia da classe fornecida com algumas alterações, dependendo do tipo de operador a ser criado. As alterações que se aplicam a todos os tipos de operadores são alteração do nome da classe, adição de um atributo e alteração do corpo do método que implementa a funcionalidade do operador. A alteração do nome da classe tem o objetivo de evitar erros causados pela tentativa de carregamento de duas classes com nomes iguais em um mesmo *class loader*. A adição de um atributo permite manter uma referência para o objeto a ser invocado pelo *proxy*. Por fim, a alteração do corpo do método que implementa a funcionalidade do operador tem o objetivo de substituir o corpo original por uma invocação ao mesmo método no objeto referenciado pelo *proxy*.

Nos casos de alterações dinâmicas nas implementações dos operadores *View* e *Aggregation* existe um problema adicional. Como esses operadores armazenam informações de estado de eventos que ocorreram anteriormente, é preciso que essas informações também sejam atualizadas para que fiquem em conformidade com a nova implementação,

o que deve ser feito para cada *statement* que está utilizando o operador no momento da atualização.

No caso das *Views*, o objetivo é reter eventos que possuam similaridades de acordo com alguns aspectos, como por exemplo eventos que possuam o valor de um atributo que representa localização dentro de uma determinada área geográfica. Esses aspectos são definidos por uma política, a qual representa a própria implementação do operador, e que determina em que condições eventos devem entrar ou sair da janela. Se essa política (ou seja, a implementação do operador) é atualizada, então é possível que eventos que estejam retidos na janela após a atualização não teriam sido selecionados para compor a janela se a política, no momento da entrada desse evento, fosse a que passa a vigorar a partir da atualização, o que representa uma inconsistência de estado. Operadores *Aggregation* normalmente atuam sobre conjuntos de valores definidos por eventos retidos por *Views*, e os operadores *Aggregation* retornam resultados baseados nesses valores. Dessa forma, a inconsistência de estado, nesse caso, é resultado do mesmo problema observado nas *Views*.

No Esper Dinâmico, atualizações de estado não foram implementadas. Porém, os principais desafios associados à essa tarefa são discutidos na seção de trabalhos futuros do Capítulo 7.

4.3.3 Camada de Gerenciamento de Eventos de Entrada

Essa camada é composta por componentes denominados *Source Adapters*, que são *bundles* OSGI responsáveis por gerar os fluxos de eventos de entrada a serem processados pelo Esper a partir da comunicação com *Sources*. Ela é formada por diversos componentes porque o conjunto de *Sources* que venha a compor um sistema de processamento de fluxos de informação pode possuir características bastante heterogêneas no que diz respeito aos protocolos de comunicação implementados pelos elementos que integram esse conjunto. Por isso, cada *Source Adapter* se comunica com um conjunto de *Sources* que implementam um protocolo de comunicação comum.

Ao receber notificações de *Sources*, esses módulos as convertem em um formato que esteja de acordo com o modelo de contexto utilizado, sendo que esse procedimento é seguido por uma invocação à API do Esper Dinâmico com o intuito de enviar as notificações recebidas por *Sources* para serem processadas pela máquina de eventos. O modelo utilizado por um componente *Source Adapter* é especificado pela entrada de cabeçalho `Require-Bundle: bundleName;bundle-version=bundleVersion`, a qual deve ser definida no arquivo `MANIFEST` contido no *bundle* referente ao *Source Adapter* em questão.

4.3.4 Camada de Gerenciamento de Eventos de Saída

Essa camada é formada pelos componentes *Sink Manager* e *Servidor Codebase*. O *Sink Manager* é um *bundle* OSGI responsável por gerenciar o registro de regras de aplicação no Esper Dinâmico e por disseminar os eventos de interesse de acordo com as regras registradas. O *Servidor Codebase* é responsável por fornecer definições de componentes de modelo para *download* sempre que *Sinks* recebem notificações que utilizam definições de componentes desconhecidas em tempo de execução. Isso é chamado de injeção de tipos em [22], e a principal finalidade é resolver dependências de tipos entre sistemas distribuídos.

O componente *Sink Manager* oferece dois serviços, sendo eles *Serviço de Registro de Interesse* e *Serviço de Recebimento de Eventos Derivados*. O primeiro é utilizado por *Sinks* para solicitar a criação de regras de aplicação no Esper Dinâmico com o intuito de registrar interesse na ocorrência de eventos derivados. O segundo é utilizado pelo Esper Dinâmico para notificar a ocorrência de eventos derivados ao componente *Sink Manager*, que é responsável por disseminar as notificações de ocorrência desses eventos para os *Sinks* que registraram interesse nos mesmos.

Essas notificações são disseminadas pelo componente *Sink* através da invocação de um serviço RMI que deve ser implementado por todos os *Sinks*. O RMI foi escolhido porque ele permite a injeção de tipos através da especificação da propriedade *codebase* [15], que indica o local remoto onde a definição de componente de modelo desconhecida pode ser encontrada. No caso da implementação proposta, esse local é o *Servidor Codebase*.

4.4 Discussão

Como dito no início do capítulo, uma implementação da arquitetura proposta para o sistema de gerenciamento de contexto deve atender os requisitos mencionados na Seção 4.1. Para a implementação proposta, o requisito *modularização dos modelos de contexto* foi atendido através da implementação de modelos como módulos OSGi que contém informações adicionais no arquivo de `MANIFEST`. Essas informações caracterizam um *bundle* como sendo um modelo de contexto gerenciável dinamicamente.

A implementação de modelos como módulos OSGi permite que o requisito *criação, atualização e remoção dinâmica de modelos de contexto* seja parcialmente atendido. Essa abordagem permite que atualizações sobre o conjunto de definições de componentes de um modelo de contexto sejam propagadas para elementos arquiteturais que utilizam o modelo atualizado. Porém, para casos de atualizações em definições de componentes de modelo já existentes, o OSGi não garante que as instâncias criadas

anteriormente às atualizações permanecerão consistentes com as novas definições. Para componentes de modelo do tipo *operadores*, a técnica de programação *proxy* é utilizada como uma abordagem complementar para atender o requisito em questão. Para os outros tipos de componentes, apenas operações de criação e remoção são tratadas.

O requisito *propagação distribuída de alterações em modelos* trata da questão da injeção de tipos em componentes arquiteturais que não operam no ambiente de execução de um SGDMC, ou seja, os *Sinks*. Alterações em um conjunto de componentes de modelo associadas aos tipos *eventos contextuais* e *entidades*, que são os tipos utilizados por *Sinks*, são propagadas, de forma distribuída, através do mecanismo *codebase* do protocolo RMI.

Por fim, o requisito *processamento ininterrupto de regras* foi obtido através da implementação de modelos como módulos OSGi, do uso de *proxies* e da representação orientada a objetos que o Esper utiliza para definir tipos de eventos, que são construções associadas aos componentes de modelo *eventos contextuais* e *entidades*. A implementação de modelos como módulos OSGi permite que novas definições de componentes de modelo sejam reconhecidas pelo Esper Dinâmico. O uso de *proxies* permite que a funcionalidade implementada por um operador seja delegada para objetos auxiliares criados dinamicamente no contexto de execução de uma regra. Por fim, a natureza orientada a objetos da representação de tipos de eventos no Esper permite que especializações desses tipos sejam reconhecidas por uma regra já em execução.

4.5 Sumário

Esse capítulo apresentou os requisitos que uma arquitetura para gerenciamento dinâmico de modelos de contexto deve atender. Além disso, foi apresentado o conceito de dependência entre modelos, o qual é utilizado na organização hierárquica implementada pelo mecanismo de gerenciamento dinâmico proposto nesse capítulo. Foram discutidos também os detalhes de implementação da infraestrutura proposta, sendo que essa discussão cobriu tópicos como o papel do OSGi nessa infraestrutura, modularização dos modelos de contexto através do OSGi e a implementação do Esper Dinâmico. Por fim, foi feita uma discussão sobre como os requisitos mencionados inicialmente foram implementados pela infraestrutura.

O próximo capítulo apresenta um estudo de caso de uma aplicação de chat, a qual é utilizada em cenários sensíveis ao contexto onde atualizações dinâmicas de modelos são necessárias. O capítulo demonstra como essas atualizações podem impactar as aplicações e a máquina IFP que gerencia os modelos utilizados por elas, além de demonstrar como a infraestrutura proposta nesse capítulo pode ser utilizada para lidar com esses impactos.

CAPÍTULO 5

Estudo de Caso

Esse capítulo apresenta um estudo de caso de implementação de um cenário de *chat* sensível ao contexto, similar ao discutido no Capítulo 2, como forma de demonstrar como a arquitetura proposta permite a manutenção dinâmica de modelos de contexto.

5.1 Descrição da Aplicação

A aplicação de *chat* é um cliente que obtém os contextos "Disponível" e "Ocupado" dos contatos a partir das informações contextuais providas pelo sistema de gerenciamento dinâmico de modelos de contexto apresentado no Capítulo 4. Esses contextos permitem a um usuário identificar o momento mais adequado para iniciar uma comunicação com um contato, e eles são determinados a partir de regras independentes da aplicação.

O contexto "Ocupado" possui um caráter dinâmico, podendo ser inferido de diversas formas. Por isso, a aplicação interage com uma máquina de eventos que avalia regras que detectam especializações de "Ocupado" que possuem uma semântica mais precisa, como por exemplo *Ocupado ministrando uma palestra*. Essas informações adicionais ajudam o usuário a avaliar se um momento é adequado para iniciar uma interação. A Figura 5.1 ilustra um protótipo de tela da aplicação de *chat* proposta.

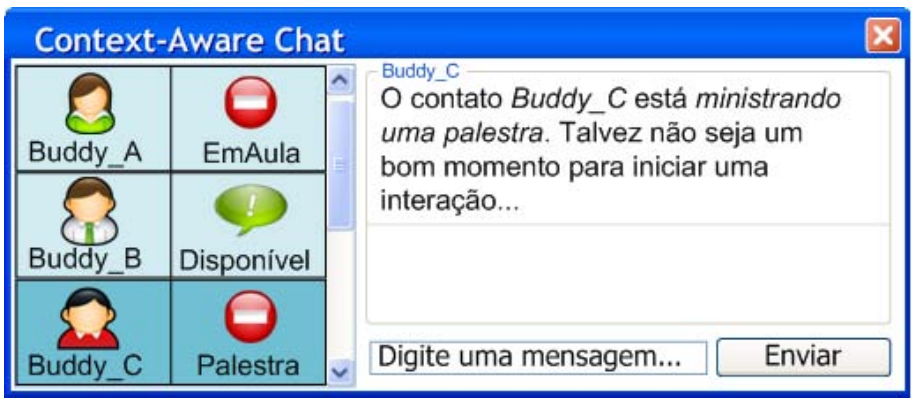
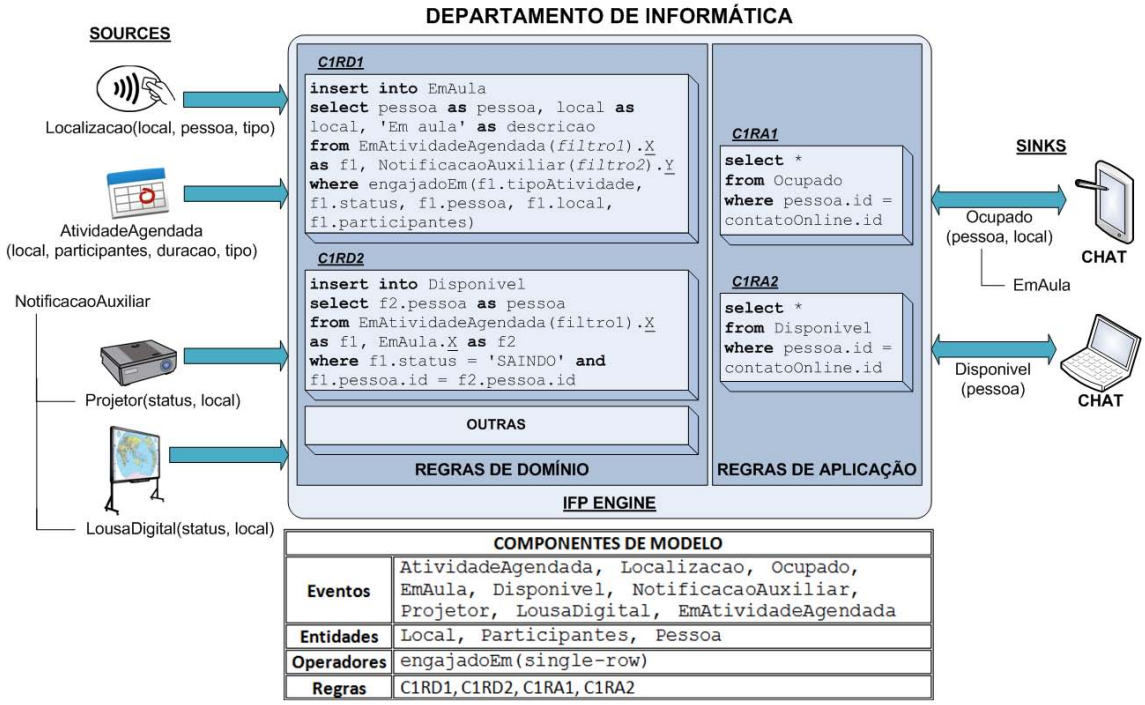


Figura 5.1: *Protótipo da aplicação de chat sensível ao contexto.*

A aplicação será utilizada em uma universidade para reduzir as interrupções indesejadas que influenciam a produtividade de alunos e professores, e aplicada em dois cenários particulares. No cenário inicial, descrito na seção 5.2, as regras de inferência verificam se professores e alunos estão participando de uma aula. No cenário seguinte, descrito na seção 5.3, o modelo de contexto inicial é modificado com o objetivo de refletir as novas situações de disponibilidade e indisponibilidade causadas pela ocorrência de um evento de pesquisa na universidade. Essas modificações ficam transparentes para a aplicação, que mantém as mesmas consultas por eventos contextuais na máquina de eventos, especificadas pelas consultas EPL `select * from Ocupado where pessoa.id = contatoOnline.id` e `select * from Disponivel where pessoa.id = contatoOnline.id`.

5.2 Primeiro Cenário

A Figura 5.2 ilustra o cenário inicial de uso do *chat*. Os eventos contextuais de indisponibilidade disseminados pela máquina de eventos e consumidos pelos *sinks* são identificados em termos de eventos de alto nível e eventos específicos recebidos de fato durante a execução do cenário. Desta maneira, de acordo com a figura, a aplicação de *chat* interpreta o evento recebido como do tipo *Ocupado*, enquanto que o evento disseminado pela máquina de eventos é *EmAula*, que é uma especialização do evento anterior.



COMPONENTES DE MODELO	
Eventos	AtividadeAgendada, Localizacao, Ocupado, EmAula, Disponivel, NotificacaoAuxiliar, Projetor, LousaDigital, EmAtividadeAgendada
Entidades	Local, Participantes, Pessoa
Operadores	engajadoEm(single-row)
Regras	C1RD1, C1RD2, C1RA1, C1RA2

Figura 5.2: *Cenário inicial de uso da aplicação de chat do departamento de informática.*

O cenário utiliza os eventos de *Localizacao* e *AtividadeAgendada* para inferir os contextos "Disponível" e "Ocupado" dos usuários. A localização é obtida através de leitores de *smartcards* instalados na entrada de cada sala do prédio do departamento. Uma leitura de localização especifica o atributo `tipo`, que indica se uma pessoa entrou ou saiu de uma sala.

Eventos do tipo *AtividadeAgendada* indicam a existência de uma atividade agendada, como por exemplo uma aula. Nesse cenário, esses eventos são obtidos a partir de um *source adapter* que se comunica com um banco de dados do departamento, o qual armazena horários e locais de aulas associados a alunos matriculados e professores designados para as aulas em questão. No momento previsto para o início de uma atividade, o *source adapter* mencionado irá gerar um evento que represente esse acontecimento, o qual descreve o local da atividade, participantes previstos, tempo de duração e tipo de atividade.

A regra $C1RD1$ utiliza os eventos *EmAtividadeAgendada*, os quais são derivados de *Localizacao* e *AtividadeAgendada*, para detectar eventos do tipo *EmAula*, que é uma especialização de *Ocupado*. Os eventos *EmAtividadeAgendada* possuem o atributo `status` para indicar se uma pessoa está entrando ou saindo desse estado. Eventos desse tipo com o `status` igual a `ENTRANDO` são geradas em duas situações:

1. quando uma pessoa entra em uma sala na qual está agendada uma atividade no momento de sua entrada e a pessoa em questão é um participante previsto da atividade;
2. quando uma pessoa já está em uma sala, um evento do tipo *AtividadeAgendada* é gerado para essa localidade e a pessoa que já estava na sala é um participante previsto.

Por outro lado, eventos do tipo *EmAtividadeAgendada* com o `status` igual a `SAINDO` são geradas nas seguintes situações:

1. quando uma pessoa sai de uma sala na qual está agendada uma atividade no momento de sua saída e a pessoa em questão é um participante previsto da atividade;
2. quando uma pessoa está em uma sala na qual está agendada uma atividade, a pessoa em questão é um participante previsto e o tempo de duração dessa atividade expira.

Essas quatro condições são especificadas pelas regras *C1RD3* e *C1RD4*, não mostradas na figura, e que serão comentadas com maiores detalhes na Seção 5.4.

Eventos do tipo *EmAtividadeAgendada* não são produzidos por *sources* e nem consumidos por *sinks*. Eles são utilizados pela máquina de eventos, na regra $C1RD1$, como um indício de que uma pessoa está *EmAula*. Apesar de ser um bom indício de que uma pessoa está engajada em alguma atividade, esse tipo de evento, derivado de informações

de atividades agendadas e da localização de participantes previstos, nem sempre pode ser utilizado como uma fonte confiável e definitiva de que a pessoa em questão está, de fato, engajada na atividade detectada.

Por isso, a regra $C1RD1$ utiliza os eventos contextuais $Projetor$ e $LousaDigital$[1] juntamente com o operador $engajadoEm$, conforme ilustrado na figura. Esse operador determina se uma pessoa está, de fato, engajada em uma atividade e, para isso, leva em conta outros fatores particulares ao evento em questão.

No cenário descrito nessa seção, esse operador leva em conta informações de *status* (ligado ou desligado) de projetores e lousas digitais presentes em uma sala para a qual existe um evento do tipo $EmAtividadeAgendada$ associado. Essas informações são coletadas pelo operador $engajadoEm$ como uma forma de garantir que as pessoas que se encontram no estado $EmAtividadeAgendada$, do tipo $AULA$, estão, de fato, participando de uma aula.

De uma forma abstrata, a implementação desse operador para o tipo de atividade $AULA$ pode ser descrito da seguinte maneira: se uma pessoa se encontra no estado $EmAtividadeAgendada$ (`status=ENTRANDO`), com `tipoAtividade=AULA` (condição especificada pela variável `filtro1` ilustrada na figura), e o projetor ou a lousa digital do local onde essa pessoa se encontra estão ligados, então a pessoa está, de fato, engajada na atividade $AULA$. A detecção dessa situação pelo operador $engajadoEm$ causa a produção de um evento do tipo $EmAula$, conforme especifica a regra $C1RD1$. Nessa regra, os elementos $\underline{X}$ (também usado em $C1RD2$) e $\underline{Y}$ representam a utilização de outras janelas de forma aninhada. O elemento $\underline{X}$ retém, para avaliação da regra, os os últimos eventos do tipo $EmAtividadeAgendada$ gerados para cada cliente (`std:groupwin(pessoa.id).std:lastevent()`), enquanto $\underline{Y}$ retém apenas o último evento do tipo $NotificacaoAuxiliar$.

O principal aspecto a se notar é que $engajadoEm$ é um operador de semântica variável, dependente da atividade realizada pela pessoa, e que, por isso, pode ser implementado de maneira distinta em cada um dos possíveis subtipos de $AtividadeAgendada$.

A regra $C1RD2$ detecta eventos do tipo $Disponivel$, os quais ocorrem quando uma pessoa entra no estado $EmAtividadeAgendada$ com `status=SAINDO` e existia um evento do tipo $EmAula$ associado a essa pessoa. As regras $C1RA1$ e $C1RA2$ são utilizadas por aplicações para registrar interesse, respectivamente, nos estados $EmAula$ e $Disponivel$. Para cada usuário *online*, a aplicação solicita a criação das regras $C1RA1$ e $C1RA2$ para a máquina de eventos. Por outro lado, quando um contato fica *offline*, uma solicitação é feita para que as regras correspondentes sejam removidas.

[1] A variável `filtro2` ilustrada na figura restringe os fluxos $NotificacaoAuxiliar$ a serem avaliados pela regra $C1RD1$ aos subtipos $Projetor$ e $LousaDigital$)

5.3 Segundo Cenário

Considere agora que o departamento de informática sedie um evento de pesquisa, para o qual alguns professores assumiram responsabilidades como coordenar palestras e sessões técnicas em laboratórios. Como esse cenário é particular e temporário, é natural que o modelo de contexto anterior não descreva adequadamente as novas condições de disponibilidade e indisponibilidade de professores associadas ao evento. Entretanto, para que a mesma aplicação de *chat* continue sendo relevante e consistente nesse novo cenário, o modelo de contexto deverá ser modificado, passando a conter componentes adicionais que descrevam situações de participação nas atividades do evento.

A Figura 5.3 ilustra as alterações realizadas no modelo de contexto decorrentes do novo cenário. Todas as regras do cenário anterior foram mantidas, e duas novas regras foram acrescentadas para inferência dos contextos *CoordenandoPA* e *CoordenandoST*, que indicam, respectivamente, uma pessoa coordenando uma palestra e uma sessão técnica. Além disso, foram acrescentadas duas novas regras de disponibilidade.

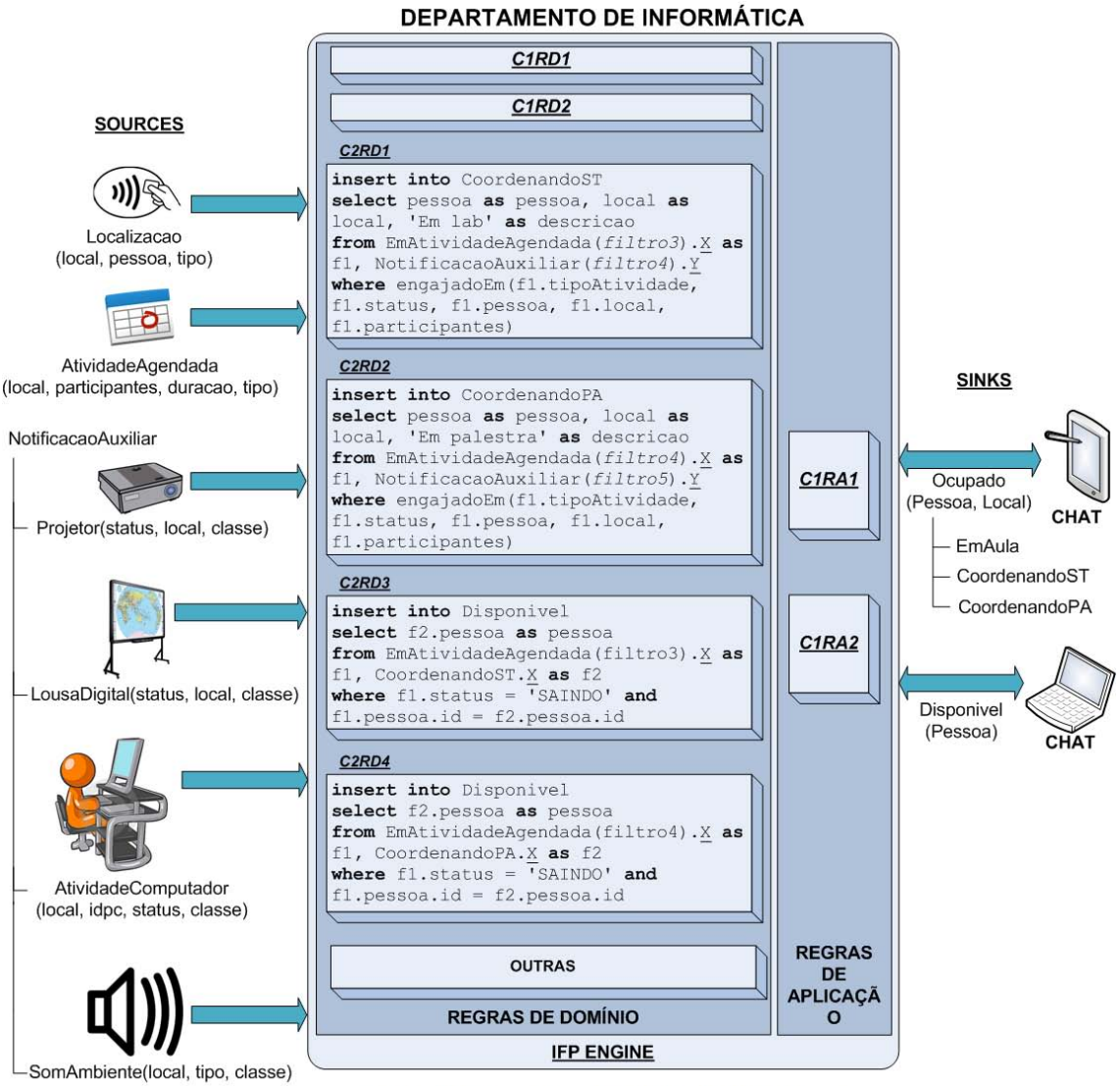
O novo cenário adiciona dois *sources*, os quais publicam eventos dos tipos *AtividadeComputador* e *SomAmbiente*. O *source* associado ao tipo *AtividadeComputador* publica informações de uso dos computadores localizados nos laboratórios onde estão previstas ocorrências das atividades de sessão técnica. Essas informações são obtidas através de aplicativos que foram instalados nesses computadores. Esses aplicativos avaliam constantemente o uso do teclado ou do mouse e enviam notificações periódicas que classificam um computador como $ATIVO$ ou $INATIVO$.

O *source* associado ao tipo $SomAmbiente$ funciona de forma semelhante ao sistema descrito em [20], o qual é capaz de detectar padrões de sons em ambientes através do uso de amostras previamente coletadas e de técnicas de aprendizado. Esses provedores avaliam os sinais emitidos por sensores de som que foram instalados nas salas onde estão previstas ocorrências de palestras.

As regras $C2RD1$ e $C2RD2$ são utilizadas, respectivamente, para produzirem eventos dos tipos *CoordenandoST* e *CoordenandoPA*, e elas funcionam de forma semelhante à regra $C1RD1$ descrita na seção anterior. As principais diferenças são os filtros utilizados para limitar os eventos dos tipos $EmAtividadeAgendada$[2] e $NotificacoesAuxiliares$[3], e a implementação do operador $engajadoEm$, que foi alterada para contemplar os novos tipos de atividade.

Para o caso da atividade de coordenação de palestra, o operador $engajadoEm$ verifica informações de classificação do som ambiente para determinar se uma palestra está, de fato, ocorrendo. Se a palestra está ocorrendo e o coordenador designado está

[2] `tipoAtividade=SESSAO_TECNICA` para $C2RD1$ e `PALESTRA` para $C2RD2$
[3] $AtividadeComputador$ para $C2RD1$ e $SomAmbiente$ para $C2RD2$



COMPONENTES DE MODELO CRIADOS/ATUALIZADOS	
Eventos	AtividadeComputador, SomAmbiente, CoordenandoST, CoordenandoPA
Entidades	-
Operadores	engajadoEm(single-row)
Regras	C2RD1, C2RD2, C2RD3, C2RD4

Figura 5.3: *Novo cenário com modelo de contexto atualizado.*

presente, então o mesmo deve estar no estado *CoordenandoPA*. Para o caso da atividade de coordenação de sessão técnica, o operador *engajadoEm* verifica informações de *status* de atividade de computadores para verificar se uma sessão técnica está ocorrendo. Se uma quantidade aceitável[4] de computadores de um determinado laboratório estiverem com o *status* ativo e o coordenador designado estiver presente no laboratório em questão, então o mesmo deve estar no estado *CoordenandoST*.

[4]Esse valor é calculado com base na quantidade de participantes previstos.

5.4 Implementação

5.4.1 Implementação do primeiro cenário

A Figura 5.4 apresenta um diagrama que ilustra como *sources*, regras e *sinks*, referentes ao primeiro cenário, se relacionam em termos de produção e consumo de eventos. Esse diagrama é baseado na notação adotada pelos autores de [11], onde as setas que entram em componentes indicam o consumo de eventos, enquanto as setas que saem de componentes indicam a produção de eventos.

A regra $C1RD3$ produz eventos do tipo *EmAtividadeAgendada* com o atributo `status=ENTRANDO`. Os eventos produzidos por $C1RD3$ são consumidos pela regra $C1RD1$ e mantidos em uma janela que retém os últimos eventos do tipo *EmAtividadeAgendada*, com `tipoAtividade=AULA`, por pessoa. Sempre que um evento desse tipo estiver presente nessa janela, com `status=ENTRANDO`, e um evento do tipo $NotificacaoAuxiliar$ (subtipos $Projetor$ ou *LousaDigital*) for consumido pela regra, o operador $engajadoEm$ será invocado. O objetivo dessa invocação é verificar se as pessoas associadas aos eventos do tipo *EmAtividadeAgendada* mantidos pela janela estão, de fato, engajadas na atividade `AULA`, o que ocorre quando o projetor ou a lousa digital do local onde essas pessoas se encontram estão ligados. Caso essa verificação se confirme, um evento *EmAula* é produzido.

Na Figura 5.4, a verificação mencionada é ilustrada pelas consultas que o operador $engajadoEm$ realiza nas janelas *StatusLousaDigital* e $StatusProjetor$. Essas janelas armazenam o último evento enviado por cada $Projetor$ e por cada *LousaDigital*, e consultar essas janelas é equivalente a solicitar informações de *status* diretamente aos projetores e lousas digitais. Essa consulta indireta realizada pelo operador $engajadoEm$ ocorre porque não é tarefa do operador conhecer os protocolos de comunicação implementados por *sources*.

Para evitar que outros subtipos de $NotificacaosAuxiliar$ sejam consumidos pela regra $C1RD1$, é utilizado um filtro que verifica o atributo `classe` de cada evento do tipo $NotificacaosAuxiliar$. Na regra $C1RD1$, esse filtro é representado pela variável do tipo `List` *classesNotificacoesAuxiliares*[5], que inicialmente contém valores que representam as classes de eventos auxiliares dos tipos $Projetor$ e *LousaDigital*. Como a regra $C1RD1$ especifica interesse nesses eventos através do evento de alto nível associado a eles ($NotificacaosAuxiliar$), é possível que novas informações contextuais sejam consideradas para refinar a detecção de eventos do tipo *EmAula* sem que, para isso, a regra

[5]O operador `verificaPresenca`, que utiliza a variável `classesNotificacoesAuxiliares`, simplesmente verifica se um determinado valor está presente em uma lista.

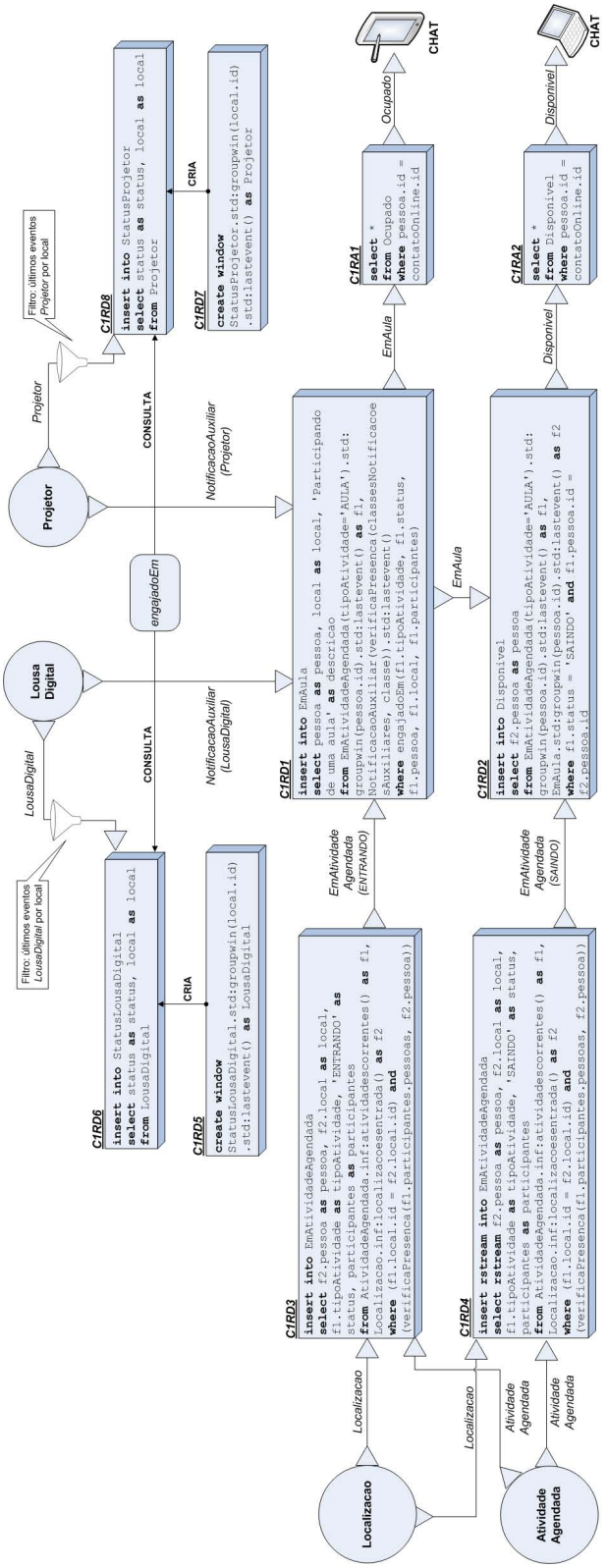


Figura 5.4: *Diagrama de produção e consumo de eventos referente ao primeiro cenário.*

$C1RD1$ tenha que ser parada ou modificada. Para tanto, bastaria que os seguintes passos fossem seguidos:

1. criar e instalar os *source adapters* que se comunicam com os novos provedores de informações contextuais;
2. ajustar o filtro (adicionar os novos tipos de eventos contextuais) que especifica as classes de eventos do tipo *NotificacaoAuxiliar* que devem ser consideradas pela regra;
3. criar novas tabelas que armazenam eventos das novas classes de *NotificacaoAuxiliar*;
4. alterar a implementação do operador *engajadoEm* para considerar o uso dos valores especificados nessas novas tabelas.

5.4.2 Implementação do Segundo Cenário

A Figura 5.5 apresenta um diagrama que ilustra como *sources*, regras e *sinks*, referentes ao segundo cenário, se relacionam em termos de produção e consumo de eventos. Nesse diagrama, foram adicionadas, com relação ao diagrama anterior, duas regras que produzem novos tipos de eventos de indisponibilidade, sendo elas *C*2*RD*1, que produz *CoordenandoST*, e *C*2*RD*2, que produz *CoordenandoPA*. Além disso, foram adicionadas as correspondentes regras de disponibilidade (*C*2*RD*3 e *C*2*RD*4). Por fim, foram adicionados dois *sources* que produzem eventos do tipo *NotificacaoAuxiliar* (subtipos *AtividadeComputador* e *SomAmbiente*) e quatro regras que criam e populam as janelas que são consultadas pela nova implementação do operador *engajadoEm* (*C*2*RD*5, *C*2*RD*6, *C*2*RD*7 e *C*2*RD*8).

Essa implementação atualizada do operador *engajadoEm* considera dois novos tipos de atividade, sendo elas `ST` (sessão técnica) e `PA` (palestra). Dessa forma, quando o parâmetro `tipoAtividade` passado para *engajadoEm* for `ST`, a janela *StatusAtividadeCmp* será consultada como forma de garantir que a pessoa associada ao evento *EmAtividadeAgendada* está, de fato, coordenando uma sessão técnica. Quando o parâmetro `tipoAtividade` for `PA`, a janela *StatusSomAmbiente* será consultada. Por fim, a implementação continua válida para o cenário anterior, ou seja, quando `tipoAtividade` for `AULA`, as janelas *StatusLousaDigital* e *StatusProjetor* é que serão consultadas.

5.4.3 Implementação dos modelos como *bundles* OSGi

As Listagens 5.1 e 5.2 mostram os arquivos `MANIFEST` que descrevem os *bundles* correspondentes aos modelos do primeiro e do segundo cenário. O segundo modelo declara dependência com relação ao primeiro através do cabeçalho `Require-Bundle: br.ufg.inf.modeloinicial`.

No cenário inicial, apenas o *bundle* da Listagem 5.1 está instalado. Para o segundo cenário, além da instalação do *bundle* da Listagem 5.2, o primeiro *bundle* deverá

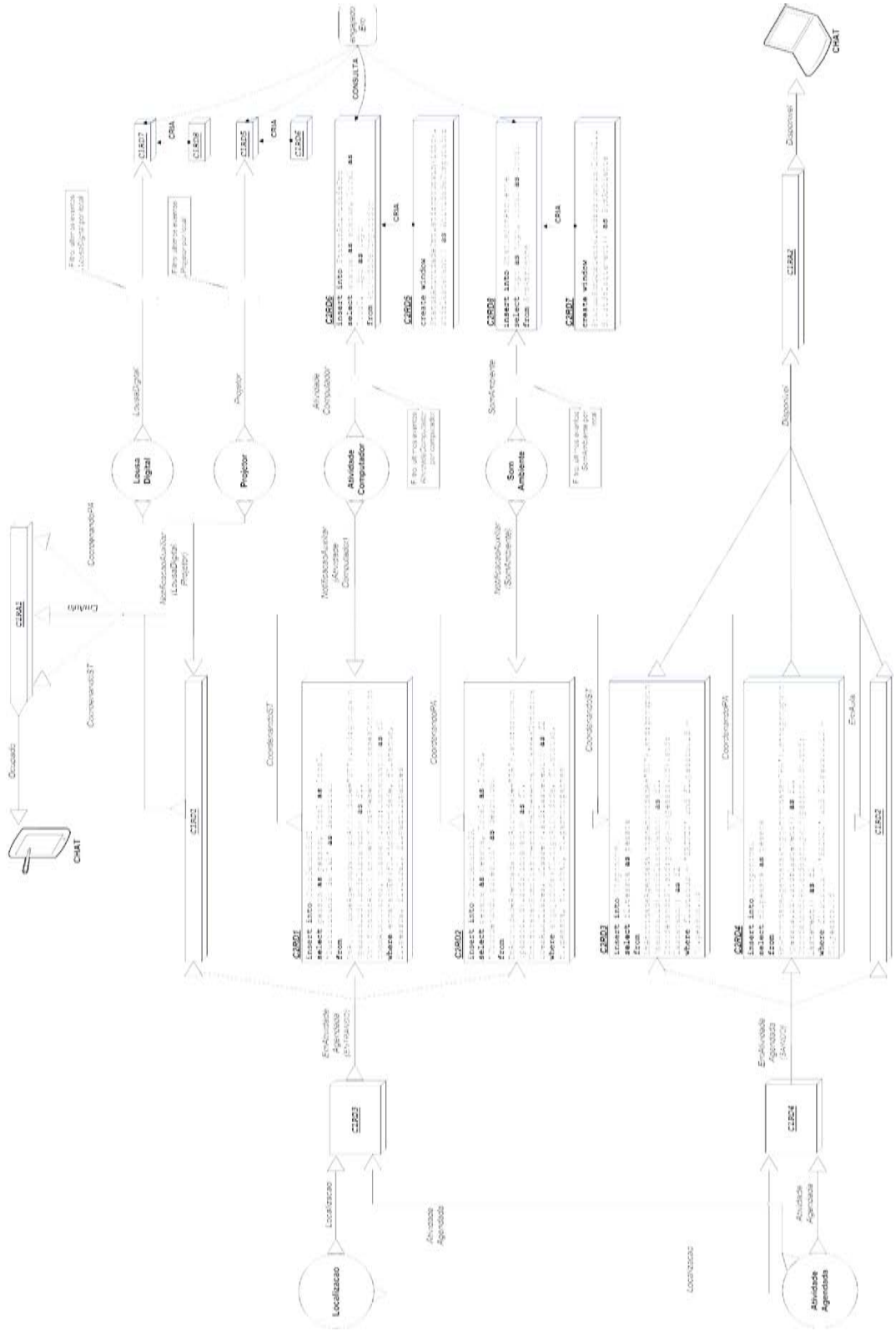


Figura 5.5: *Diagrama de produção e consumo de eventos referente ao segundo cenário.*

ser atualizado para contemplar as alterações no operador *engajadoEm*. Para tanto, a linha `Bundle-Version` do `MANIFEST` correspondente deverá ser alterada para 1.0.1, indicando

a presença de uma nova versão. Além disso, a linha `singlerow[engajadoEm...]` deverá ser alterada para especificar a classe que contém a nova implementação do operador.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: ModeloInicial
Bundle-SymbolicName: br.ufg.inf.modeloinicial
Bundle-Version: 1.0.0
Export-Package: br.ufg.inf.modeloinicial.entities,
 br.ufg.inf.modeloinicial.events,
 br.ufg.inf.modeloinicial.events.auxiliar,
 br.ufg.inf.modeloinicial.operators
 ...
Bundle-Type: ContextModel
ContextModel-EventTypes:
 Ocupado;br.ufg.inf.modeloinicial.events.Ocupado,
 EmAula;br.ufg.inf.modeloinicial.events.EmAula,
 Disponivel;br.ufg.inf.modeloinicial.events.Disponivel,
 Projetor;br.ufg.inf.modeloinicial.events.auxiliar.Projetor,
 ...
ContextModel-Rules:
 C1RD1;insert into EmAula...,
 C1RD2;insert into Disponivel...,
 ...
ContextModel-Operators:
 singlerow[engajadoEm;engajadoEm;br.ufg.inf.modeloinicial.operators.OperadoresSingleRow],
 ...
```

Listagem 5.1: *`MANIFEST` referente ao modelo do cenário inicial.*

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: SegundoModelo
Bundle-SymbolicName: br.ufg.inf.segundomodelo
Bundle-Version: 1.0.0
Export-Package: br.ufg.inf.segundomodelo.events,
 br.ufg.inf.segundomodelo.events.auxiliar
Require-Bundle: br.ufg.inf.modeloinicial
 ...
Bundle-Type: ContextModel
ContextModel-EventTypes:
 CoordenandoPA;br.ufg.inf.segundomodelo.events.CoordenandoPA,
 CoordenandoST;br.ufg.inf.segundomodelo.events.CoordenandoST,
 AtividadeComputador;br.ufg.inf.segundomodelo.events.auxiliar.AtividadeComputador,
 SomAmbiente;br.ufg.inf.segundomodelo.events.auxiliar.SomAmbiente
ContextModel-Rules:
 C2RD1;insert into CoordenandoST...,
 C2RD2;insert into CoordenandoPA...,
 C2RD3;insert into Disponivel...,
 C2RD4;insert into Disponivel...
ContextModel-Operators:
```

Listagem 5.2: *`MANIFEST` referente ao modelo do segundo cenário.*

5.5 Discussão

O estudo de caso apresentado demonstra a necessidade de suporte da arquitetura proposta para manutenção dinâmica de modelos em cenários de computação sensível ao contexto distribuídos e dinâmicos. Os cenários apresentam atualizações temporárias no modelo de contexto, mas mesmo atualizações permanentes podem causar problemas em aplicações, caso elas não utilizem uma infraestrutura equivalente à proposta.

A primeira consequência do cenário é a necessidade de reconhecimento de especializações dos componentes de modelo dos tipos entidades e eventos em aplicações. Embora tal reconhecimento seja comum em linguagens que permitem polimorfismo de tipos, no ambiente distribuído é necessário o reconhecimento dos novos componentes (entidades e eventos) para que as aplicações se comportem consistentemente em tempo de execução. No cenário, isso ocorre quando eventos *EmAula*, *CoordenandoST* e *CoordenandoPA* são disseminados para as aplicações de *chat* que estão preparadas para lidar com eventos *Ocupado*, só para citar uma das situações geradas pelo cenário. A arquitetura lida com esses problemas utilizando um protocolo de distribuição baseado em Java RMI em conjunto com um servidor de *codebase* para carga dinâmica dos novos componentes de modelos. Com Java RMI, o carregamento de componentes de modelo é implementado como carregamento dinâmico de classes.

A segunda consequência do cenário é a necessidade de atualização dinâmica do operador *engajadoEm*. A modelagem adotada no cenário permite uma construção incremental e adaptativa das condições que levam a uma situação de indisponibilidade. Em geral, tais situações não podem ser adequadamente previstas e dependem de uma série de condições em tempo de execução. Como a *máquina IFP* utiliza *proxies* de operadores, uma operação pode ser direcionada para elementos externos à própria arquitetura, como uma outra máquina de inferência baseada em regras, tornando a solução arquitetural híbrida com relação ao método de inferência oferecido.

Por fim, o cenário *pode* gerar uma situação de inconsistência de eventos entre o modelo inicial e o atualizado. Considere, por exemplo, o caso em que um professor esteja coordenando uma palestra em uma sala para a qual existe uma atividade de `AULA` e outra de `CP` agendadas para esse professor. Nesse caso, os eventos *EmAula* e *CoordenandoPA* poderiam ser gerados ao mesmo tempo.

Uma forma de resolver esse problema seria atualizar o operador de semântica variável *atividadescorrentes*. Esse operador é utilizado nas regras que produzem *EmAtividadeAgendada*, e ele implementa uma *view* que retém todos os eventos do tipo *AtividadeAgendada* do momento X até $X + duracao$, onde X é o momento em que esses eventos são recebidos pelo *esper* e *duracao* é o tempo previsto de duração da atividade correspondente. Uma possível atualização desse operador para evitar as inconsistências

mencionadas seria verificar se já existe uma atividade agendada para o mesmo local e com horários em comum. Entretanto, isso depende de uma modelagem cuidadosa do operador por parte do *rule manager*. Por esse motivo, uma arquitetura deveria oferecer mecanismos para lidar com tais conflitos e inconsistências, o que não é objetivo desse trabalho.

5.6 Limitações

Um primeiro aspecto limitante a se notar com relação aos cenários propostos nesse estudo de caso é que todos os eventos gerados por *sources* são simulados. Dessa forma, o conjunto de situações detectadas pelas regras de domínio são limitadas pela ocorrência de um conjunto restrito de eventos simulados.

Um outro aspecto limitante é que todas as informações contextuais consumidas pelo operador $engajadoEm$ devem ser subtipos de $NotificacaoAuxiliar$. Essa restrição pode criar problemas nas situações em que um determinado evento que não é do tipo $NotificacaoAuxiliar$ passe a ser de interesse do operador $engajadoEm$. Isso ocorre porque a implementação da arquitetura proposta não fornece suporte para atualizações que envolvam alterações em relações hierárquicas.

Por fim, todas as situações de disponibilidade ou indisponibilidade discutidas nos cenários possuem como pré-requisito que a tarefa associada esteja previamente agendada, o que não ocorre na maioria dos casos.

5.7 Sumário

Esse capítulo apresentou um estudo de caso de implementação de um cenário de *chat* sensível ao contexto. O *chat* implementado foi utilizado em duas situações, sendo que a segunda exigiu modificações no modelo de contexto utilizado na primeira. Apesar das modificações, foi demonstrado que as atualizações não causaram inconsistências nos modelos utilizados pelas aplicações.

Além disso, esse capítulo demonstrou como a infraestrutura para gerenciamento dinâmico de modelos proposta no Capítulo 4 lida com os impactos causados por atualizações dinâmicas sobre componentes de modelo, os quais foram discutidos na Seção 2.6. Por fim, as principais limitações do estudo de caso foram comentadas.

O próximo capítulo apresenta os principais trabalhos relacionados com os temas tratados nessa dissertação.

CAPÍTULO 6

Trabalhos Relacionados

O gerenciamento dinâmico de modelos de contexto é um assunto pouco explorado na literatura em modelos para computação sensível ao contexto. As pesquisas nesta área podem ser classificadas em (i) desenvolvimento de sistemas baseados em eventos (IFP) ou (ii) abordagens para modelagem de contexto. Enquanto que o primeiro foca na implementação de consultas complexas e disseminação aos clientes, o segundo foca na construção de modelos que ofereçam expressividade e permitam modelar situações complexas, especialmente modelos baseados em ontologias. As pesquisas geradas nessas duas sub-áreas devem lidar com um trade-off em eficiência na disseminação, programabilidade e expressividade dos modelos. Como resultado, muitos trabalhos têm explorado a adoção de modelos de contexto híbridos.

Foi discutido nesse trabalho a atualização dinâmica de modelos, que é um problema que envolve tanto as máquinas de gerenciamento de contexto, como as abordagens de modelagem de contexto. De fato, quanto mais simples for a abordagem de modelagem de contexto, mais simples também é a solução do problema atacado. Por exemplo, abordagens que utilizam modelos mais simples como par-valor, permitem naturalmente a atualização dinâmica dos modelos, mas em contra-partida dificultam o desenvolvimento das aplicações complexas, assim como a modelagem de situações complexas.

Nesse trabalho, foi escolhida a implementação Esper de CEP como estudo de caso, com o objetivo de selecionar uma abordagem que oferecesse um mínimo de expressividade dos modelos e uma rica linguagem para construção de consultas. De fato, a atualização dinâmica impõe dificuldades para gerenciamento de modelos e funcionamento das máquinas de eventos em tempo de execução.

Abordagens de modelagem baseadas em ontologias oferecem a construção de modelos dinâmicos. Entretanto, tais abordagens não oferecem um mecanismo de disseminação de eventos contextuais, que deve ser implementado como um componente externo ao gerenciador de ontologias. Como resultado, embora os modelos possam ser dinâmicos, os dinamismo não é capturado nas consultas feitas pelas aplicações, que não podem lidar transparentemente com essas atualizações.

Entre os trabalhos em máquinas de disseminação de eventos, [12] propõe uma so-

lução arquitetural para oferecer modificação nos comportamentos das máquinas de eventos, incluindo o modelo de eventos adotado e a linguagem de consulta. Entretanto, a abordagem permite apenas a variabilidade estática, ou seja, a mudança no comportamento das máquinas apenas antes da sua execução. Outros trabalhos que lidam com a modificação no comportamento das máquinas são FULCRUM [4] e o trabalho proposto em [14].

O trabalho [7] lida com o problema de atualização dinâmica de modelos. Entretanto, o trabalho oferece soluções para um ambiente distribuído, no qual a mudança nos modelos é um efeito da mobilidade entre domínios, enquanto que este trabalho foca no efeito das atualizações nas máquinas de eventos, quando consideradas isoladamente. Os protocolos inter-domínios implementados naquele trabalho podem ser incorporados neste trabalho, assim como a arquitetura interna do sistema de gerenciamento de contexto poderia ser incorporada naquele. Neste sentido, os dois trabalhos são complementares.

O trabalho [24] também lida com o problema de atualização dinâmica de modelos. Porém, o objetivo principal se limita a permitir a incorporação dinâmica de novos tipos de *entidades* e *eventos contextuais*, que são chamados de *observables*. Instâncias de *observables* são produzidas por sistemas chamados *collectors*, e uma mesma instância do *middleware* proposto no trabalho mencionado pode se comunicar com *collectors* com diferentes APIs e que publicam diferentes tipos de *observables*. Essa comunicação é feita através de *bridges* [13], que são os elementos criados dinamicamente para permitir a comunicação com *collectors* que não foram previstos durante a fase de implementação de um sistema baseado no *middleware*.

Os modelos de contexto utilizados em [24] são baseados em um meta-modelo, o qual é utilizado como base para a criação de *entidades* e *observables* associados a um cenário sensível ao contexto. Além desses dois elementos, o meta-modelo também define *contratos*, que são utilizados para especificar, por exemplo, o modelo de comunicação (síncrono ou assíncrono) entre o *middleware* e um *collector*. Os modelos são especificados utilizando a ferramenta EMF [23], que permite a geração dinâmica de classes a partir de modelos estruturados descritos com XMI[1].

Uma vantagem desse trabalho com relação ao trabalho [24] são as atualizações dinâmicas dos operadores utilizados em uma regra e a criação dinâmica de regras, as quais são definidas em tempo de compilação em [24]. Além disso, a modularização dos modelos de contexto implementada nesse trabalho permite que diferentes tipos de aplicação sejam criadas com base na infraestrutura proposta. Em [24], o uso de um modelo de contexto atômico limita o uso do *middleware* a um tipo específico de aplicação.

[1] XML Metadata Interchange - padrão para troca de informações baseado em XML

CAPÍTULO 7

Conclusões

Esse trabalho apresentou uma infraestrutura para gerenciamento dinâmico de modelos de contexto, a qual permite a adequação de modelos, em tempo de execução, a cenários de computação sensível ao contexto com características dinâmicas e imprevisíveis, como o cenário apresentado no estudo de caso. Além disso, a infraestrutura permite o desenvolvimento de vários tipos de aplicações sensíveis ao contexto que compartilham modelos de contexto organizados de forma hierárquica.

A infraestrutura mencionada adota um modelo de sistema baseado no funcionamento dos *Sistemas de Processamento de Fluxos de Informação*, ou IFPS, que é um termo utilizado pelos autores de [19] para caracterizar sistemas que possuem como principal requisito o processamento contínuo de fluxos de informação. Dentre as tecnologias de IFPS, o CEP, através do uso da máquina de processamento de eventos Esper, foi selecionado como estudo de caso para implementação da infraestrutura proposta.

Para permitir a adequação de modelos a cenários dinâmicos, são necessárias atualizações, em tempo de execução, sobre componentes de modelo, que são as abstrações fundamentais a partir das quais um modelo é construído. Porém, alguns tipos de atualização podem gerar inconsistências simbólicas, que são problemas que podem exigir o redesenvolvimento de uma aplicação para que ela possa utilizar modelos atualizados de maneira consistente. Como a proposta desse trabalho é permitir atualizações em tempo de execução (ou seja, sem a necessidade de redesenvolvimento), as atualizações suportadas pela infraestrutura proposta foram limitadas àquelas que se enquadram na categoria de atualização suave, as quais não provocam os problemas de inconsistência mencionados.

Atualizações suaves sobre componentes de modelo podem provocar impactos sobre o funcionamento das máquinas IFP que os gerenciam e sobre as aplicações sensíveis ao contexto que os utilizam. Os principais impactos analisados foram o reconhecimento distribuído de tipos, a produção simultânea de eventos semanticamente inconsistentes e a influência de atualizações em modelos sobre os mecanismos de *matching* e *filtering* de uma máquina IFP. Um estudo de caso foi apresentado para demonstrar a ocorrência desses impactos em um cenário dinâmico e temporário. As atualizações realizadas sobre os modelos referentes ao estudo de caso demonstraram que os mecanismos implementados

pela infraestrutura proposta são capazes de lidar com os impactos mencionados.

7.1 Contribuições

A principal contribuição desse trabalho é a infraestrutura proposta para gerenciamento dinâmico de modelos de contexto, a qual é baseada nas tecnologias Esper e OSGi.

Por ser baseada em Esper, a infraestrutura proposta representa um rico estudo de caso da aplicação dos conceitos de IFPS em computação sensível ao contexto. O Esper implementa uma linguagem de processamento de fluxos que está entre as mais expressivas e completas dentre as tecnologias avaliadas pelos autores de [19]. Dessa forma, os fundamentos utilizados no projeto e implementação da infraestrutura apresentada nesse trabalho podem ser aproveitados na construção de sistemas baseados em outros tipos de máquinas IFP.

Outra contribuição dessa dissertação é a proposta de implementação de modelos de contexto como módulos OSGi. Essa proposta auxilia na construção e manutenção dinâmica de uma hierarquia de modelos através do conceito de dependência entre modelos. Isso é de fundamental importância para suportar o desenvolvimento de diversos tipos de aplicações sensíveis ao contexto com diferentes requisitos no que diz respeito ao nível de abstração dos componentes descritos pelos modelos de contexto a serem utilizados.

7.2 Trabalhos Futuros

Como trabalhos futuros, destacamos a necessidade de expandir o conjunto de operações de atualização suave sobre componentes de modelos de contexto, que nesse trabalho foram limitadas a inclusão de novos eventos e entidades, inclusão de novas regras e atualização de implementação de operadores de semântica variável através de *proxies*.

Nesse trabalho, as operações de atualização suave sobre componentes de modelo eventos contextuais e entidades, que são implementados como classes Java, foram limitadas por restrições da especificação atual da Máquina Virtual Java. Essa especificação não permite que a definição de uma classe seja recarregada ao longo da execução de um aplicativo. Essa limitação evita, por exemplo, que um novo atributo seja adicionado dinamicamente a uma classe. Para atingir esse objetivo, a definição da classe deveria ser alterada para contemplar o novo atributo. Em seguida, a definição atualizada deveria ser recarregada pela máquina virtual, o que não é possível de acordo com a especificação mencionada.

Porém, existem alguns trabalhos com propostas para superar essa limitação, como é o caso da Máquina Virtual *Dynamic Code Evolution* [27], a qual permite redefi-

nições ilimitadas de classes em tempo de execução. A análise e utilização de propostas de trabalhos como esse podem auxiliar na meta de expandir o conjunto de operações de atualização suave sobre componentes de modelos de contexto.

Com relação ao Esper Dinâmico, destacamos a necessidade de implementar suporte para atualizações dinâmicas do operador *pattern*, o que é importante para permitir que regras avaliem condições mais complexas, como por exemplo detecção de sequências pré-definidas de eventos. Além disso, destacamos a necessidade de implementar atualização de estado nos operadores de semântica variável *Views* e *Aggregations*, conforme discutido na Seção 4.3.2. Uma das principais dificuldades associada à essa melhoria é que a atualização de estado desses operadores exige a manipulação e alteração dos dados mantidos por todos os *statements* registrados no Esper que utilizam os operadores atualizados.

Referências Bibliográficas

[1] ALLIANCE, O. **OSGi Service Platform, Core Specification, Release 4, Version 4.2**. Technical report, OSGI Alliance, Sept. 2009.

[2] BETTINI, C.; BRDICZKA, O.; HENRICKSEN, K.; INDULSKA, J.; NICKLAS, D.; RANGANATHAN, A.; RIBONI, D. **A survey of context modelling and reasoning techniques**. *Pervasive and Mobile Computing*, 6(2):161 – 180, 2010. Context Modelling, Reasoning and Management.

[3] BODOFF, S.; ARMSTRONG, E.; BALL, J.; CARSON, D. B. **The J2EE Tutorial, Second Edition**. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2004.

[4] BOYER, R. T.; GRISWOLD, W. G. **Fulcrum - an open-implementation approach to internet-scale context-aware publish / subscribe**. *Hawaii International Conference on System Sciences*, 9:275a, 2005.

[5] BUCHHOLZ, T.; KRAUSE, M.; LINNHOFF-POPIEN, C.; SCHIFFERS, M. **CoCo: Dynamic composition of context information**. In: *First Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services (MobiQuitous'04)*, p. 335–343, 2004.

[6] CHIBA, S.; NISHIZAWA, M. **An easy-to-use toolkit for efficient java bytecode translators**. In: *Proceedings of the 2nd international conference on Generative programming and component engineering*, GPCE '03, p. 364–376, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[7] DA ROCHA, R. C. A. **Context Management for Distributed and Dynamic Context-Aware Computing**. PhD thesis, Departamento de Informatica, Pontificia Universidade Catolica do Rio de Janeiro, Rio de Janeiro, 2009.

[8] DEY, A. K. **Understanding and using context**. *Personal and Ubiquitous Computing*, 5:4–7, 2001. 10.1007/s007790170019.

[9] DOCKHORN COSTA, P.; ANDRADE ALMEIDA, J. P.; FERREIRA PIRES, L.; GUIZZARDI, G.; VAN SINDEREN, M. J. **Towards conceptual foundations for context-aware**

applications. In: Roth-Berghofer, T. R.; Schulz, S.; Leake, D. B., editors, *Modeling and Retrieval of Context. Papers from the 2006 AAAI Workshop, Boston, Massachusetts*, volume WS-06-12 de **AAAI Technical Report**, p. 54–58, Menlo Park, California, 2006. AAAI Press.

[10] ESPERTECH INC.. **Esper Reference Documentation**, 2011. Version 4.2.0.

[11] ETZION, O.; NIBLETT, P. **Event Processing in Action**. Manning Publications Co., 2010.

[12] FILHO, R. S. S.; DE SOUZA, C. R. B.; REDMILES, D. F. **The design of a configurable, extensible and dynamic notification service**. In: *Proceedings of the 2nd international workshop on Distributed event-based systems*, DEBS '03, p. 1–8, New York, NY, USA, 2003. ACM.

[13] GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. **Design patterns: elements of reusable object-oriented software**. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[14] GAZZOTTI, M.; MAMEI, M.; ZAMBONELLI, F. **A programmable event-based middleware for pervasive mobile agent organizations**. *Parallel, Distributed, and Network-Based Processing, Euromicro Conference on*, 0:517, 2003.

[15] GROSSO, W. **Java RMI**. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1st edition, 2001.

[16] HALL, R.; PAULS, K.; MCCULLOCH, S.; SAVAGE, D. **OSGi in Action: Creating Modular Applications in Java**. Manning Pubs Co Series. Manning, 2011.

[17] HENRICKSEN, K.; INDULSKA, J. **Developing context-aware pervasive computing applications: Models and approach**. *Pervasive and Mobile Computing*, 2(1):37–64, February 2006.

[18] MAGID, Y.; SHARON, G.; ARCUSHIN, S.; BEN-HARRUSH, I.; RABINOVICH, E. **Industry experience with the ibm active middleware technology (amit) complex event processing engine**. In: *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, DEBS '10, p. 140–149, New York, NY, USA, 2010. ACM.

[19] MARGARA, A.; CUGOLA, G. **Processing flows of information: from data stream to complex event processing**. In: *ACM Computing Surveys*, 2012. (to appear).

[20] NEGISHI, Y.; KAWAGUCHI, N. **Instant learning sound sensor: Flexible environmental sound recognition system**. In: *Networked Sensing Systems, 2007. INSS '07. Fourth International Conference on*, p. 305, june 2007.

[21] RANGANATHAN, A.; CAMPBELL, R. H.; RAVI, A.; MAHAJAN, A. **ConChat: A context-aware chat program**. *IEEE Pervasive Computing*, 1:51–57, 2002.

[22] RELLERMEYER, J. S.; ALONSO, G.; ROSCOE, T. **R-osgi: distributed applications through software modularization**. In: *Proceedings of the 8th ACM/IFIP/USENIX international conference on Middleware*, MIDDLEWARE2007, p. 1–20, Berlin, Heidelberg, 2007. Springer-Verlag.

[23] STEINBERG, D.; BUDINSKY, F.; PATERNOSTRO, M.; MERKS, E. **EMF: Eclipse Modeling Framework**. Addison-Wesley, 2nd edition, 2009.

[24] TACONET, C.; KAZI-AOUL, Z.; ZAIER, M.; CONAN, D. **Ca3m: A runtime model and a middleware for dynamic context management**. In: *Proceedings of the Confederated International Conferences, CoopIS, DOA, IS, and ODBASE 2009 on On the Move to Meaningful Internet Systems: Part I*, OTM '09, p. 513–530, Berlin, Heidelberg, 2009. Springer-Verlag.

[25] TRUONG, B. A.; LEE, Y.-K.; LEE, S.-Y. **Modeling and reasoning about uncertainty in context-aware systems**. In: *e-Business Engineering, 2005. ICEBE 2005. IEEE International Conference on*, p. 102 –109, oct. 2005.

[26] WANG, X.; ZHANG, D.; GU, T.; PUNG, H. **Ontology based context modeling and reasoning using owl**. In: *Pervasive Computing and Communications Workshops, 2004. Proceedings of the Second IEEE Annual Conference on*, p. 18 – 22, march 2004.

[27] WÜRTHINGER, T.; WIMMER, C.; STADLER, L. **Dynamic code evolution for java**. In: *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, PPPJ '10, p. 10–19, New York, NY, USA, 2010. ACM.