Auto-manutenção de Classes de Fusão em Visões de Integração de Dados

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Valéria magalhães Pequeno e aprovada pela Banca Examinadora.

Fortaleza, 28 de Abril de 2000.

Profa. Dra. Vânia M. Ponte Vidal
(Orientadora)

Dissertação apresentada ao Mestrado em Ciência da Computação, UFC, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Mestrado em Ciência da Computação
Universidade Federal do Ceará

Auto-manutenção de Classes de Fusão em Visões de Integração de Dados

Valéria magalhães Pequeno[1]

28 de Abril de 2000

Banca Examinadora:

- Profa. Dra. Vânia M. Ponte Vidal (Orientadora)
- Prof. Dr. Rubens de Nascimento Melo
- Prof. Dr. Ângelo Roncalli A. Brayner
-

[1] Ex-bolsista da FUNCAP e da CAPES

© Valéria magalhães Pequeno, 2002.
Todos os direitos reservados.

Prefácio

Dados de múltiplos bancos de dados podem ser integrados através de visões que são materializadas e armazenadas em um banco de dados. Uma vantagem de tal sistema é que as consultas do usuário podem ser respondidas diretamente, não necessitando do acesso às fontes de informação remotas. Sua principal desvantagem é que as visões materializadas devem ser atualizadas (mantidas) a fim de ficarem consistentes com os bancos de dados locais.

Um tipo de classe que ocorre com freqüência em visões de integração de dados são as chamadas *classes de fusão*. Nesse tipo de classe, os objetos são obtidos da fusão de vários objetos dos bancos de dados locais que representam o mesmo objeto do mundo real. Isso significa que se dois objetos são semanticamente equivalentes então estes são sintetizados em um único objeto da classe de fusão. O problema com classes de fusão é que estas nem sempre são auto-manuteníveis, ou seja, é necessário acessar os bancos de dados locais para fazer sua manutenção. A auto-manutenção é uma característica importante para as visões de integração, uma vez que pode ser muito caro consultar os bancos de dados remotos.

Neste trabalho, propomos uma estratégia para permitir a auto-manutenção, de forma incremental, de *classes de fusão completas* em visões materializadas. Esse tipo de classe é mantido através de regras locais, as quais propagam para a visão, as atualizações a serem realizadas na classe de fusão completa, sem a necessidade de acessarem as bases de dados locais.

Outra contribuição deste trabalho é o suporte à auto-manutenção de outros tipos de classe de fusão, utilizando para isso as classes de fusão completas como classes auxiliares. A extensão da classe de visão original é um subconjunto da extensão da classe auxiliar correspondente, podendo ser definida virtualmente como uma simples seleção usando apenas essa classe auxiliar. A classe de visão original também pode ser implementada como uma visão materializada. Nesse caso, devem ser geradas regras que propagam atualizações da classe auxiliar em atualizações da classe de visão. Isso é feito de forma a tornar a classe de visão consistente com a classe auxiliar e por conseguinte com relação aos bancos de dados locais.

Uma característica importante do enfoque proposto é que as regras para fazer a

manutenção das classes de visão são geradas a partir das assertivas de correspondência. Essas assertivas são tipos especiais de restrições de integridade que especificam formalmente o relacionamento entre o esquema da visão e os esquemas locais. Sua utilização evita o uso de uma álgebra de objetos específica e permite a geração de regras corretas.

Conteúdo

Prefácio iv

1 Introdução 1

1.1 Integração de Dados Usando Visões 1

1.2 Manutenção de Visões Materializadas 4

1.3 Solução Proposta para a Auto-manutenção de Classes de Fusão 5

2 Manutenção de Visões Materializadas 8

2.1 Introdução 8

2.2 Políticas de Manutenção de Visões Materializadas 9

2.3 Estratégias de Manutenção de Visões Materializadas 10

2.3.1 Rematerialização 10

2.3.2 Manutenção Incremental 11

2.4 Regras ECA e suas Semânticas de Execução 12

2.5 Utilizando Regras para a Manutenção de Visões Materializadas 14

2.6 Trabalhos Relacionados 17

2.6.1 Manutenção de Visões Materializadas em Ambientes Centralizados . 18

2.6.2 Manutenção de Visões Materializadas em Ambientes Distribuídos . 19

3 Visões no Modelo de Objetos 22

3.1 Introdução 22

3.2 O Modelo de Objetos 23

3.3 Assertivas de Correspondência 27

3.3.1 Assertivas de Correspondência de Extensão 27

3.3.2 Assertivas de Correspondência de Propriedades 29

3.3.3 Assertivas de Correspondência de Caminhos 29

3.4 Visões de Integração de Dados e Classes de Fusão 31

3.4.1 Definindo Classes de Fusão 34

3.5 Usando as Assertivas de Correspondência para Definir Classes de Visão . . 36

4 Auto-Manutenção de Classes de Fusão Completas **39**
4.1 Introdução 39
4.2 Classes de Fusão Completas 40
4.3 Enfoque Proposto para a Auto-Manutenção de Classes de Fusão Completas 42
4.4 Gerando as Regras para a Manutenção das Classes de Fusão Completas . . 45
4.4.1 Gerando Regras para a Manutenção Incremental de Visões a partir das Assertivas de Correspondência 46
4.4.2 Gerando as Regras para a Manutenção das Classes de Fusão Completas a partir das Assertivas de Correspondência 47

5 Usando Classes de Fusão Completas para Suportar a Auto-manutenção de Classes de Visão que Preservam os Objetos **66**
5.1 Introdução 66
5.2 A Arquitetura Proposta para a Auto-Manutenção de Classes de Visão . . . 67
5.3 Definindo Classes Auxiliares 68
5.4 Definindo Classes de Visão a partir das Classes Auxiliares 72
5.4.1 Classes de Visão Originais Virtuais 73
5.4.2 Classes de Visão Originais Materializadas 74
5.5 Gerando Regras para Manter Classes de Visão a partir das Classes Auxiliares 75
5.5.1 Manutenção das ACEs de Equivalência 76
5.5.2 Manutenção das ACEs de Diferença 76
5.5.3 Manutenção das ACEs de União 77
5.5.4 Manutenção das ACEs de Interseção 78
5.5.5 Manutenção das ACEs de Seleção 79

6 Conclusões **81**

Bibliografia **85**

A Algoritmos **91**
A.1 Algoritmo A_1 91
A.2 Algoritmo A_2 92

Lista de Tabelas

2.1 Características dos Trabalhos Relacionados e do Enfoque Proposto 21

3.1 Assertivas de Correspondência de Extensão 28

5.1 Mapeamento entre Assertivas de Correspondência de Extensão 72

Lista de Figuras

1.1 Enfoque virtual para a integração de dados 2
1.2 Enfoque materializado para a integração de dados 3

2.1 A arquitetura adotada no projeto *Squirrel* [13] 15

3.1 Exemplo de um Esquema OO 25
3.2 Caminhos "semanticamente equivalentes" 30
3.3 Exemplo de classes que usam operações de junção 30
3.4 Esquemas locais $\mathbf{S}_1$ e $\mathbf{S}_2$ 32
3.5 Esquema S_v da visão $\mathbf{V}$ 33
3.6 Especificação da classe de visão EST&EMP na linguagem MSL [46] 35
3.7 Esquema da visão S_v 37

4.1 Regra para manutenção da classe de visão ALUNO_v nas inserções em ESTUDANTE 39
4.2 Esquema da visão S_v com as ACs entre EST&EMP e suas classes raízes . . . 41
4.3 Arquitetura proposta para a auto-manutenção de classes de fusão completas 42
4.4 Regra para manutenção de Ψ_k nas remoções em $\mathbf{C}_k$ 48
4.5 Assertivas de Correspondência entre EST&EMP e suas classes raízes 49
4.6 Regra para manutenção de Ψ_2 nas remoções em EMPREGADO 49
4.7 Método *GereEIO_de_*EST&EMP 50
4.8 Método *RemovaTipo* 51
4.9 Regra para manutenção de Ψ_k nas inserções em $\mathbf{C}_k$ 51
4.10 Regra para manutenção de Ψ_2 nas inserções em EMPREGADO 52
4.11 Método *GereEO_de_*EST&EMP 52
4.12 Método *AdicioneTipo* 54
4.13 Método *CrieObjeto* 55
4.14 Regra para manutenção de Ψ:$\mathbf{F}.\boldsymbol{p} \equiv \mathbf{C}_k.\boldsymbol{p'}$ onde $\boldsymbol{p}$ e $\boldsymbol{p'}$ são atributos . . . 56
4.15 Regra para a manutenção de Ψ_6 nas atualizações em $\textbf{salário}_1$ 57
4.16 Regra para manutenção de Ψ:$\mathbf{F}.\boldsymbol{p} \equiv \mathbf{C}_k.\boldsymbol{p'}$ onde $\boldsymbol{p}$ e $\boldsymbol{p'}$ são relacionamentos 57
4.17 Regra para a manutenção de Ψ_7 nas atualizações em $\textbf{curso}_2$ 58

4.18 Regra para manutenção de Ψ nas inserções em **c**.***p’***, onde ***p’*** é um atributo 59
4.19 Regra para a manutenção de Ψ_8 nas inserções em **c.telefone**$_2$ 59
4.20 Regra para manutenção de Ψ nas remoções em **c**.***p’***, onde ***p’*** é um atributo 59
4.21 Regra para manutenção de Ψ_8 nas remoções em **c.telefone**$_2$ 60
4.22 Regra para manutenção de Ψ nas inserções em **c**.***p’***, onde ***p’*** é um relacionamento 61
4.23 Regra para manutenção de Ψ_{11} nas inserções em **c.dependentes**$_1$ 61
4.24 Regra para manutenção de Ψ nas remoções em **c**.***p’***, onde ***p’*** é um relacionamento 62
4.25 Regra para manutenção de Ψ_{11} nas remoções em **c.dependentes**$_1$ 62
4.26 Regra para manutenção de Ψ nas modificações em **r**.$\boldsymbol{p}_i$, onde $\boldsymbol{p}_i$ é uma das propriedades de um caminho de valor 63
4.27 Regra para manutenção de Ψ_{12} nas modificações em **d.nome**$_1$ 64
4.28 Regra para manutenção de Ψ nas modificações em **r**.$\boldsymbol{p}_i$ onde $\boldsymbol{p}_i$ é uma das propriedades de um caminho de referência 65

5.1 Arquitetura proposta para a auto-manutenção de classes de visão 67
5.2 Esquemas locais S_1 e S_2 69
5.3 Esquema de visões S_{v1} e S_{v2} com ACs 70
5.4 Classe auxiliar EST&EMP para a classe de visão ALUNO$_v$ 71
5.5 Classe auxiliar EST&EMP para as classes de visão ALUNO$_v$ e EMPREGADO$_v$ 71
5.6 Definição da classe de visão EMPREGADO$_v$ 73
5.7 Regra para manutenção de Ψ': **V**≡**A**[“**C**” ∈ **classes**] nas remoções em **a.classes** 76
5.8 Regra para manutenção de Ψ':**V**≡**A**[“**C**” ∈ **classes**] nas inserções em **a.classes** 76
5.9 Regra para manutenção de Ψ': **V**≡**A**[“**C**$_1$” ∈ **classes** ∧ “**C**$_2$” ∉ **classes**] nas remoções em **a.classes** 77
5.10 Regra para manutenção de Ψ': **V**≡**A**[“**C**$_1$” ∈ **classes** ∧ “**C**$_2$” ∉ **classes**] nas inserções em **a.classes** 77
5.11 Regra para manutenção de Ψ':**V**≡**A**[“**C**$_1$” ∈ **classes** ∨ ... ∨ “**C**$_m$” ∈ **classes**] nas remoções em **a.classes** 78
5.12 Regra para manutenção de Ψ':**V**≡**A**[“**C**$_1$” ∈ **classes** ∨ ... ∨ “**C**$_m$” ∈ **classes**] nas inserções em **a.classes** 78
5.13 Regra para manutenção de Ψ':**V**≡**A**[“**C**$_1$” ∈ **classes** ∧ ... ∧ “**C**$_m$” ∈ **classes**] nas remoções em **a.classes** 78
5.14 Regra para manutenção de Ψ':**V**≡**A**[“**C**$_1$” ∈ **classes** ∧ ... ∧ “**C**$_m$” ∈ **classes**] nas inserções em **a.classes** 79

5.15 Regra para manutenção de Ψ':**V**≡**A**[**p** ∧ "**C**" ∈ **classes**] nas remoções em **a.classes** . . . 79
5.16 Regra para manutenção de Ψ':**V**≡**A**[**p** ∧ "**C**" ∈ **classes**] nas inserções em **a.classes** . . . 80
5.17 Regra para manutenção da ACE Ψ':**V**≡**A**[**p** ∧ "**C**" ∈ **classes**] nas modificações em **a**.***p*** . . . 80

6.1 Ferramenta para a definição de classes de visão e das regras e operações para mantê-las . . . 83

Capítulo 1

Introdução

1.1 Integração de Dados Usando Visões

Os rápidos avanços tecnológicos na comunicação de dados e a tendência a descentralização têm impulsionado o desenvolvimento de aplicações que necessitam acessar, de forma integrada, informações distribuídas em múltiplos bancos de dados. Tais aplicações devem fornecer aos seus usuários uma visão global e consistente dos dados, de forma que estes não tomem conhecimento dos bancos de dados locais que foram acessados para fornecer as informações requisitadas. O acesso integrado a informações, armazenadas em múltiplos bancos de dados distribuídos e heterogêneos, constitui-se um dos principais desafios para os sistemas de informação modernos.

O uso de visões tem sido reconhecido como uma técnica importante para a integração de informações heterogêneas e distribuídas em múltiplos bancos de dados. Diversas arquiteturas, propostas para fornecer a interoperabilidade entre bancos de dados, utilizam visões integradas como interfaces através das quais usuários podem acessar diferentes bancos de dados. Neste trabalho, o termo *visão de integração* é utilizado para designar um esquema de banco de dados, cujos elementos (por exemplo: relações ou classes) são obtidos a partir da integração de elementos de um ou mais esquemas de bancos de dados já especificados. O termo *sistema global* é usado para referenciar o ambiente da visão de integração.

Basicamente, existem dois enfoques para suportar visões de integração: um que utiliza visões virtuais e o outro que usa visões materializadas. No enfoque virtual, as informações são extraídas das fontes de informação (bases de dados locais) somente quando requisitadas. No enfoque materializado, as informações de maior interesse de cada base de dados local são extraídas, integradas e armazenadas em um repositório.

No enfoque virtual, o processo de extração das informações pode ser resumido em

dois passos: 1) as consultas submetidas ao sistema são decompostas em subconsultas a serem submetidas às fontes de informação; 2) os resultados obtidos das subconsultas são filtrados, transformados e combinados, sendo a resposta resultante enviada ao usuário ou às aplicações do usuário [1, 2, 3, 4].

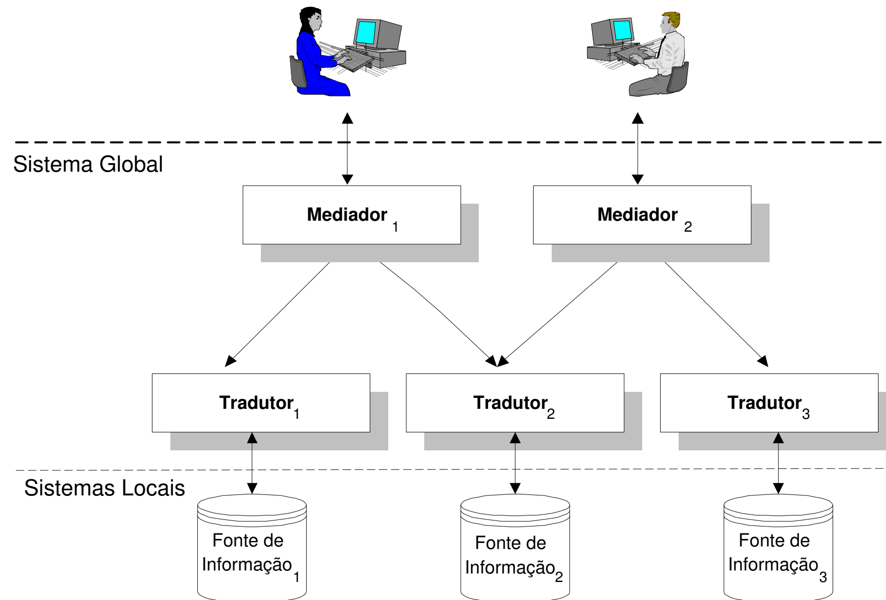

Figura 1.1: Enfoque virtual para a integração de dados

A Figura 1.1 mostra uma arquitetura de mediadores [5] que adota o enfoque virtual. Nessa arquitetura, os *mediadores* são “interfaces” através das quais os usuários consultam e atualizam múltiplas bases de dados locais [6]. Os *tradutores* são responsáveis pela transformação dos dados das fontes de informação (representação, sistema de medidas, ...) em um modelo comum e pela conversão das consultas das aplicações em consultas específicas da base de dados local correspondente. Com o uso dos mediadores, um esquema para a visão de integração é disponibilizado a partir do mediador e consultas são realizadas junto a esse esquema. Como os dados continuam armazenados apenas nas fontes de informação, diz-se que a visão de integração é virtual.

O enfoque virtual é adequado quando as informações das bases de dados locais mudam freqüentemente e os usuários realizam consultas *had-oc* constantemente. A grande desvantagem desse enfoque é que as informações têm que ser extraídas das bases de dados locais sempre que uma consulta for realizada na visão virtual. Isso pode ser muito ineficiente, especialmente quando uma mesma consulta é realizada várias vezes e quando os passos de transformação, filtragem e combinação dos dados exigem um processamento considerável [7].

No enfoque materializado, os dados são previamente integrados e armazenados em um repositório de dados, ao contrário do que ocorre na abordagem virtual, onde é preciso recuperar e integrar os dados a cada consulta efetuada. Nesse caso, a visão de integração é considerada materializada, visto que os dados não são armazenados apenas nas fontes de informação. As consultas submetidas ao sistema são então avaliadas diretamente nesse repositório, sem a necessidade de acessar as bases de dados locais [8, 9, 10].

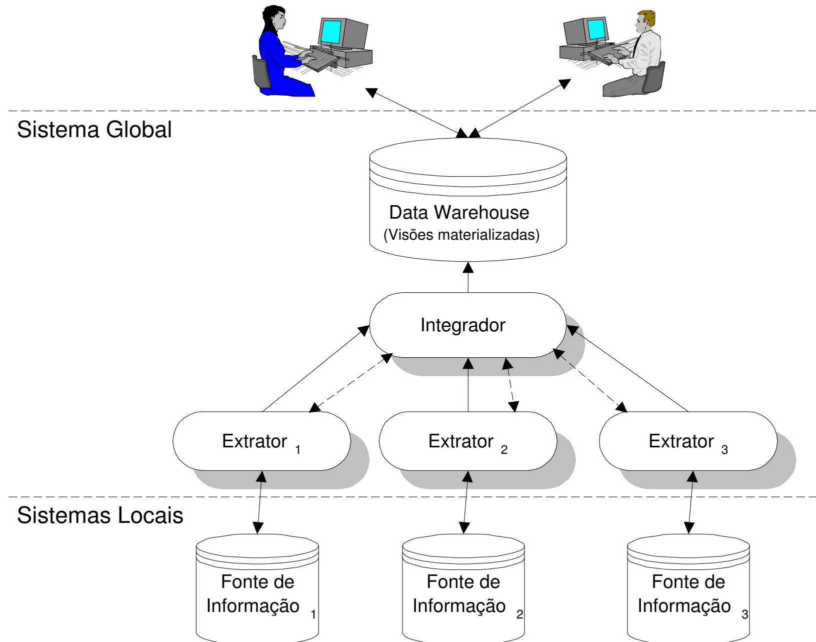

Figura 1.2: Enfoque materializado para a integração de dados

A Figura 1.2 mostra o exemplo de uma arquitetura que utiliza o enfoque materializado para integração de dados adotada no projeto *WHIPS* [11]. Nessa arquitetura o *integrador* é o módulo responsável, entre outras tarefas, por fazer a integração dos dados oriundos de várias bases de dados locais e resolver possíveis inconsistências. Os *extratores*, além de transformarem os dados em um modelo de dados comum, são responsáveis por detectar automaticamente as modificações relevantes das bases de dados e reportá-las para o integrador. O *data warehouse* é o repositório de dados do sistema onde as visões materializadas ficam armazenadas.

O enfoque materializado é adequado em aplicações, como "*data warehouse*", onde respostas rápidas são importantes e onde as fontes de informações locais nem sempre podem estar disponíveis. Além disso, visões materializadas são usadas quando não há necessidade de informações atuais.

O principal problema com o uso de visões materializadas é que estas precisam ser

periodicamente atualizadas em resposta a mudanças ocorridas nas bases de dados locais, a fim de manter a consistência entre os dados materializados e os dados das fontes de informação locais. O processo de atualização de uma visão materializada em resposta a mudanças das bases de dados locais é denominado de *manutenção da visão materializada*.

Com relação as visões virtuais e materializadas, certamente, existem situações onde uma abordagem é mais adequada do que a outra. Contudo, para aplicações mais complexas e em grande escala, ambas abordagens devem ser usadas (*enfoque híbrido*). Para manipular tais aplicações, é mais vantajoso ter parte das informações armazenadas em um repositório de dados do sistema, enquanto outras informações devem ser buscadas apenas quando consultas são requisitadas.

Uma das decisões mais importantes no enfoque híbrido é a seleção de quais visões, ou que partes da visão, devem ser materializadas. O objetivo é selecionar as visões a serem materializadas de modo a alcançar a melhor combinação entre o *bom desempenho das consultas* e *o baixo custo de manutenção das visões*. Esses dois fatores são conflitantes: o desempenho ótimo das consultas pode ser alcançado armazenando no repositório de dados do sistema o resultado de todas as consultas de interesse, em contrapartida, a manutenção das visões materializadas, nesse caso, pode tornar-se um processo árduo.

1.2 Manutenção de Visões Materializadas

Existem basicamente duas estratégias para fazer a manutenção de visões materializadas: a rematerialização e a manutenção incremental. Na rematerialização, todos os dados da visão são novamente obtidos em tempos pré-estabelecidos. Na manutenção incremental [12, 13], periodicamente apenas parte dos dados da visão são modificados em resposta às atualizações das bases de dados locais. Um dos fatores determinantes da estratégia de manutenção a ser utilizada é a capacidade das fontes de informação de notificar atualizações para o sistema global de modo que as visões materializadas possam ser atualizadas apropriadamente.

Algumas abordagens propostas para a manutenção incremental [12, 13] usam regras de produção, também denominadas regras ECA, para automaticamente atualizar as visões materializadas. Regras de produção têm associadas condições de disparo que quando satisfeitas causam uma ação a ser executada. Em [13], são usados dois tipos de regras: as regras locais e as globais. As regras locais são armazenadas nos bancos de dados locais locais e são usadas para notificar para o sistema global as atualizações ocorridas nas bases de dados locais. As regras globais são armazenadas no sistema global e são responsáveis por manterem a consistência dos dados materializados em relação as mudanças ocorridas nas bases de dados locais. Em alguns tipos de visões, pode ser necessário que o sistema

global envie consultas às bases de dados locais a fim de obter dados adicionais para determinar as atualizações que devem ser realizadas para manter corretamente a visão materializada. Em um ambiente distribuído, é preferível que as visões possam ser mantidas utilizando apenas as informações das visões e as informações enviadas pelas regras locais, isto é, sem que seja necessário enviar consultas adicionais aos bancos de dados locais locais. Uma visão que pode ser mantida dessa forma é denominada de *auto-manutenível* [14]. A auto-manutenção é uma característica importante para as visões de integração, uma vez que pode ser muito caro ou até mesmo impossível consultar os bancos de dados locais remotos, pois nem sempre estes estão disponíveis. Além disso, pode ocorrer o problema das chamadas *anomalias de atualização* [9]. Como os bancos de dados locais locais e a visão estão sendo atualizados ao mesmo tempo, pode acontecer que o resultado de uma consulta do sistema global a um banco de dados locais local corresponda a um estado do banco de dados locais diferente do estado que este tinha no momento em que regra local notificou a atualização para o sistema global.

O presente trabalho focaliza o problema da auto-manutenção, de forma incremental, de classes de fusão. *Classes de fusão* correspondem a um tipo especial de "*outerjoin view*" no modelo relacional, que são estas definidas por um "*natural outer-equijoin nos atributos chaves*" [15]. Em uma classe de fusão, os dados referentes a um dado objeto da classe de visão podem estar armazenados de forma distribuída, possivelmente com informações replicadas, em vários bancos de dados locais. Nesse caso, os objetos da classe são obtidos da fusão de vários objetos dos bancos de dados locais, os quais representam a mesma entidade do mundo real. Esse tipo de classe tem recentemente ganho importância pois são muito comuns em visões de integração. Um problema com as classes de fusão, como será mostrado neste trabalho, é que na maioria dos casos não são auto-manuteníveis.

1.3 Solução Proposta para a Auto-manutenção de Classes de Fusão

No presente trabalho, é apresentada uma estratégia para realizar a auto-manutenção, de forma incremental, de um tipo especial de classes de fusão que são as *classes de fusão completas*, as quais correspondem a visões relacionais definidas por um "*natural full outer-equijoin nos atributos chaves*". As principais características do enfoque proposto são:

- Usa o modelo de objetos para a modelagem do esquema da visão e dos esquemas locais. Nesse modelo, um esquema da visão consiste de um conjunto de definições de classes, denominadas *classes de visão*.

- Usa regras de produção para permitir a auto-manutenção de classes de fusão completas. Essas regras, denominadas regras locais, propagam para o sistema global as atualizações a serem realizadas em uma classe de fusão completa. Assim, caso informações adicionais sejam necessárias para fazer a manutenção da classe de fusão completa, estas já podem ser extraídas das fonte de informação e enviadas para o sistema global.

- As regras para fazer a manutenção das classes de fusão completas são definidas a partir das assertivas de correspondência, as quais especificam formalmente o relacionamento entre o esquema da visão e os esquemas locais. O uso das assertivas de correspondência permite provar formalmente que as regras geradas no enfoque proposto fazem corretamente a manutenção da visão. Isso significa que as atualizações produzidas pelos bancos de dados locais refletem exatamente o efeito necessário para que a visão fique consistente com os bancos de dados locais.

As classes de fusão completas podem ser utilizadas como classes auxiliares para permitir a auto-manutenção de outros tipos de classes de fusão mais genéricos. No presente trabalho, as principais características para esse enfoque são:

- As classes de visão tratadas são estas que *preservam os objetos*. Uma classe de visão preserva os objetos quando existe um mapeamento 1-1 entre os objetos da classe de visão e os objetos das classes dos bancos de dados locais, onde esses objetos representam uma mesma entidade do mundo real.

- As classes auxiliares contêm os objetos da classe de visão original bem como os objetos que são membros de classes que podem ter objetos em comum com a classe de visão (uma vez que estes objetos podem "vir a ser" inseridos na classe de visão). Esses objetos da classe auxiliar são armazenados já em uma forma "*sintetizada*", isto é, as informações de todos os objetos dos bancos de dados locais que representam uma mesma entidade do mundo real e que são membros (de fato ou em potencial[1]) das classes de visão originais são sintetizadas em um único objeto da classe auxiliar.

- Uma vez que as classes auxiliares são materializadas, uma classe de visão original pode ser definida virtualmente como uma simples seleção usando apenas a classe auxiliar correspondente. A classe de visão original também pode ser implementada como uma visão materializada. Nesse caso, devem ser geradas regras que propagam atualizações da classe auxiliar em atualizações da classe de visão original, de forma

[1]Um objeto é membro em potencial de uma classe de visão caso esse objeto não seja ainda membro dessa classe de visão, mas existe a possibilidade de que possa vir a ser inserido na classe de visão. No presente trabalho, essa semântica é capturada pelas assertivas de correspondência.

a tornar a classe de visão original consistente com a classe auxiliar e por conseguinte com relação aos bancos de dados locais.

- A definição da classe de visão original virtual e a geração das regras para as classes de visão originais materializadas são obtidas com base nas assertivas de correspondência, que especificam como as classes auxiliares estão relacionadas com as classes de visão originais.

O restante da dissertação é organizada como se segue.

- No capítulo 2, é discutido o problema da manutenção de visões materializadas. Nesse capítulo também são analisados alguns trabalhos mais relacionados com problema abordado nesta dissertação.

- No capítulo 3, é apresentado o modelo de objetos proposto com o formalismo necessário para expressar as assertivas de correspondência, as quais são usadas na geração das regras de produção. Nesse capítulo, são também definidas as classes de fusão e é mostrado como as assertivas de correspondência podem ser usadas para especificar classes de fusão.

- No capitulo 4, é mostrado o enfoque proposto para fazer a auto-manutenção de classes de fusão completas. Nesse capítulo são apresentadas a arquitetura proposta e as regras que devem ser geradas para permitir a auto-manutenção das classes de fusão completas.

- No capítulo 5, é apresentado como as classes de fusão completas podem ser usadas para permitir a auto-manutenção de outros tipos de classes de fusão. Nesse capítulo, é mostrado como as classes auxiliares podem ser obtidas. Além disso, apresenta as regras que devem ser geradas para permitir a auto-manutenção das classes de fusão.

- No capítulo 6, são mostradas algumas conclusões, contribuições e sugestões para trabalhos futuros.

Capítulo 2

Manutenção de Visões Materializadas

2.1 Introdução

Neste capítulo, será abordado o problema da manutenção de visões materializadas. Como mencionado no capítulo 1, em um ambiente de integração de dados que utiliza o enfoque materializado, os dados de maior interesse do usuário são extraídos das bases de dados e armazenados em um repositório de dados do sistema (o *data warehouse*). Essas visões materializadas quase sempre precisam ser atualizadas quando os dados das bases de dados forem alterados, a fim de manter a consistência entre os dados materializados e os dados das fontes de informação. Isto significa que as visões materializadas em um dado estado do *data warehouse* devem ter o mesmo conteúdo supondo que essas mesmas visões fossem processadas diretamente nas bases de dados. Como as bases de dados permanecem ativas e continuam a suportar aplicações locais, é possível que seus dados sofram atualizações constantemente. O processo de atualização de uma visão materializada em resposta a mudanças das bases de dados é chamado de *manutenção da visão materializada*.

O processo de manutenção de visões materializadas consiste de duas etapas: a propagação e a atualização. Na fase de propagação, as atualizações ocorridas nas classes bases são identificadas e repassadas ao sistema global. Na fase de atualização, a visão é efetivamente atualizada.

Duas questões fundamentais a serem consideradas no processo de manutenção das visões materializadas são: *quando* e *como* as visões materializadas devem ser atualizadas. Essas questões dizem respeito, respectivamente, a escolha da política de manutenção das visões que determina quando as visões devem ser materializadas e da estratégia que estabelece como as visões devem ser mantidas. Essas questões são discutidas a seguir nas

seções 2.2 e 2.3, respectivamente. O restante deste capítulo é organizado como se segue:

- Na seção 2.4, são mostradas algumas características de regras ECA. Essas regras têm sido utilizadas para fazer a manutenção de visões materializadas.
- Na seção 2.5, é apresentado o uso de regras ECA na manutenção de visões materializadas.
- Na seção 2.6, são apresentados alguns trabalhos relacionados ao problema tratado nesta dissertação.

2.2 Políticas de Manutenção de Visões Materializadas

Uma política de manutenção de visões materializadas determina o momento em que as visões materializadas devem ser atualizadas. A escolha de qual política será usada depende da freqüência de consulta e para que se destina a visão materializada [16]. A seguir, algumas políticas de manutenção são descritas[1]:

- **Atualização imediata**: a classe de visão é atualizada usando a mesma transação que atualiza a classe base. Esse tipo de manutenção torna caro o processo de atualização, uma vez que a transação de atualização pode ficar mais lenta. Essa política somente pode ser utilizada em um ambiente centralizado em situações tais como: uma taxa alta de consulta é esperada ou são exibidas respostas em tempo real. Entretanto, não deve haver muitas visões materializadas mantidas de modo imediato em um mesmo sistema. Isto porque a atualização tanto da classe base como das visões materializadas pode ficar prejudicada, nos casos em que muitas visões são derivadas de uma mesma classe base.
- **Atualização adiada**: a classe de visão é atualizada usando uma transação diferente da transação de atualização da classe base e sempre depois desta. Para fazer a manutenção de visões materializadas usando essa política pode ser necessário um arquivo para guardar as atualizações ocorridas nas classes base (de modo a ser usado em um período posterior). Pode-se distinguir três tipos diferentes de atualização adiada:
  - **Adiada ociosa** (*lazy deferred*): a classe de visão só é atualizada quando for consultada. Essa estratégia degrada o desempenho da consulta aplicada na classe de visão.

[1] No presente capítulo tudo é descrito usando o paradigma OO, embora a maioria das estratégias tenham sido originalmente desenvolvidas para o ambiente relacional.

- **Adiada periódica**: em tempos pré-estabelecidos, a classe de visão é atualizada em uma transação de atualização especial. Essa abordagem permite consultas rápidas e não retarda a atualização. Entretanto, consultas feitas na classe de visão podem acessar dados que não representam o estado atual das bases de dados. Essa política pode ser usada, por exemplo, em aplicações de suporte à decisão, que necessitam apenas de uma versão estável dos dados.
- **Adiada forçada**: a classe de visão é atualizada após ocorrer um número pré-estabelecido de atualizações nas classes base.

2.3 Estratégias de Manutenção de Visões Materializadas

As duas estratégias principais que determinam como as visões materializadas devem ser atualizadas são: *rematerialização* e *manutenção incremental*, ambas descritas a seguir.

2.3.1 Rematerialização

A rematerialização consiste em obter novamente todos os dados da visão materializada em certos tempos pré-estabelecidos. Nessa abordagem, uma gravação dos dados contidos nas classes base é realizada em um certo período. Esses dados são armazenados localmente ou em um local intermediário (nem nas bases de dados nem no repositório de dados do sistema). Então, é realizada uma comparação entre os dados obtidos através de uma gravação anterior e a gravação corrente, sendo que as modificações detectadas são enviadas à visão materializada. Essas modificações são enviadas normalmente através de mensagens.

Comparar arquivos contendo dados de classes inteiras a fim de detectar mudanças não é um processo barato. Alguns estudos foram feitos para diminuir os gastos com essa abordagem. Como exemplo, pode-se destacar o algoritmo desenvolvido em [17] que lê o arquivo a ser comparado apenas uma vez, ao contrário da maioria dos algoritmos usados para tratar esse problema.

A rematerialização é usada normalmente quando as bases de dados não podem ser consultadas, não possuem um arquivo de *log*, ou não têm capacidade ativa. Essa estratégia é muito utilizada em sistemas comerciais de *data warehouse* existentes.

2.3.2 Manutenção Incremental

Na maioria dos casos, é dispendioso obter novamente todos os dados da visão materializada para torná-la consistente com relação às suas origens. Freqüentemente, é mais vantajoso apenas modificar parte dos dados da visão em resposta as atualizações das bases de dados locais. Essa estratégia é denominada de manutenção incremental de visões materializadas.

Duas questões básicas precisam ser consideradas no processo de manutenção incremental de visões em múltiplas bases de dados:

1. ***Como as visões tomam conhecimento das atualizações nas fontes de informação***. A resposta para essa questão vai depender da capacidade ativa das bases de dados locais. De acordo com essa capacidade, as bases de dados locais podem ser:

    - *Suficientemente ativa*: as bases de dados locais são ditas suficientemente ativas no caso em que podem periodicamente enviar para o sistema global as atualizações ocorridas nas bases de dados desde a última transmissão. Esse envio normalmente é feito através de *triggers* baseados em eventos físicos ou mudanças de estados [7, 18, 19].
    - *Atividade restrita*: as bases de dados locais possuem atividade restrita quando não podem enviar para o sistema global as atualizações sucedidas nas bases de dados, porém, possuem a capacidade de transmitir mensagens por meio de *triggers* baseados em eventos físicos, tal como o *commit* de uma transação. Nesse caso, quando um evento físico ocorre, a base de dados local pode executar uma consulta e enviar o resultado para o sistema global. Em um caso extremo, quando a base de dados local fornece apenas *triggers* de monitoramento, *softwares* podem ser incorporados à base de dados local a fim de enviarem mensagens apropriadas ao sistema global.
    - *Nenhuma atividade*: as bases de dados possuem nenhuma atividade quando não dispõem de qualquer tipo de *trigger*. Nesse caso, consultas periódicas podem ser feitas a partir do sistema global a fim de se detectar as atualizações ocorridas nas bases de dados, ou os arquivos de *log* podem ser inspecionados de modo que essas atualizações possam ser extraídas a partir deles.

2. ***Como manter corretamente os dados materializados para refletir as mudanças das bases de dados locais***. A resposta para essa questão pode implicar em três tarefas: a) na identificação das classes de visão que tornaram-se inconsistentes devido a atualizações nas bases de dados locais, isto é, as classes de visão que foram afetadas pela atualização; b) no envio de consultas às bases de

dados locais a fim de buscar informações complementares (dados adicionais) para atualizar as classes de visão de modo apropriado; c) na atualização das classes de visão afetadas a fim de torná-las consistentes com o novo estado das bases de dados locais.

Uma abordagem bastante utilizada na manutenção incremental de visões, tanto em ambientes centralizados como em ambientes distribuídos, consiste em utilizar regras de produção para atualizar automaticamente as visões materializadas [12, 13, 20]. Uma regra de produção está associada a um evento e pode ocasionar a execução de uma ação. Assim, quando um determinado evento ocorre, a regra associada a ele é disparada e a ação correspondente é executada. Na seção 2.5, é apresentado como as regras de produção são utilizadas na manutenção da visão. A seguir, são mostrados alguns conceitos básicos sobre essas regras.

2.4 Regras ECA e suas Semânticas de Execução

Bancos de dados ativos estão surgindo como paradigma para a realização de manutenção incremental de visões materializadas [10]. Esses bancos de dados, ao contrário de bancos de dados convencionais, monitoram situações de maior interesse e quando elas ocorrem, disparam uma resposta apropriada. O comportamento desejado dos bancos de dados é expresso através de regras de produção, também denominadas de regras E-C-A (Evento-Condição-Ação). As regras são definidas e armazenadas nos bancos de dados locais e, normalmente, são gerenciadas por um mecanismo embutido nos próprios SGBDs locais.

As regras ECA são definidas em uma linguagem própria e possuem a seguinte forma:

NomeDaRegra
Quando **evento**
Se **condição**
Então **ação**

Quando um **evento** ocorre, a **condição** é avaliada junto ao banco de dados, caso a condição seja satisfeita, então a **ação** é executada.

Um ***evento*** é uma ocorrência num banco de dados, o qual pode ser *interno* ao banco de dados (por exemplo, atualizações) ou *externo* (por exemplo, interrupções de sistemas operacionais). O evento mais comum considerado em bancos de dados ativos são as modificações nos dados do banco de dados.

Uma ***condição*** é definida como um predicado ou consulta sobre os bancos de dados. A condição é satisfeita quando o predicado é verdadeiro ou quando a consulta retorna

uma resposta não vazia. Em uma regra, a condição pode ser omitida e nesse caso, a ação deve ser sempre executada quando a regra associada é disparada.

A ***ação*** da regra especifica as operações a serem realizadas quando a regra é disparada e quando a condição é satisfeita. Uma ação pode conter operações do banco de dados, procedimentos escritos numa linguagem de programação que podem ou não acessar o banco de dados, operações de transações, sinais de que um evento definido pelo usuário ocorreu e chamadas a procedimentos de aplicações.

A determinação de como as regras devem ser executadas é feita pelo modelo de execução, o qual é acoplado à linguagem de definição de regras. Cada linguagem de definição de regras possui uma semântica de execução, que determina como o processamento da regra é avaliado em tempo de execução uma vez que um conjunto de regras tenha sido definido. O processamento (ou reação [21]) de uma regra consiste em avaliar a condição de uma regra disparada e executar sua ação. De um modo geral, o seguinte algoritmo pode ser utilizado [20]:

Enquanto existem regras disparadas e nenhum erro foi detectado
Selecione uma regra disparada ***r*** *de acordo com algum critério de ordenamento*
Execute ***r****, ou seja, avalie a condição e, no caso dela ser satisfeita, execute a ação*
Se um erro ocorreu, restitua o sistema para o último estado consistente

Na semântica de execução de uma regra ECA estão inclusas questões como: a interação das regras com as operações do banco de dados e com transações que são submetidas por usuários e programas de aplicação. A atividade de bancos de dados convencionais, tais como consultas, modificações e atualizações, é que possibilita o disparo de uma regra e inicia seu processamento.

Vários fatores devem ser considerados na semântica de execução das regras, alguns dos quais são mostrados a seguir:

- O momento no qual uma regra disparada deve ser executada (*a granularidade da regra*). Uma regra pode ser disparada depois de cada objeto (ou tupla) ser modificado, ou após um conjunto inteiro de modificações especificadas por uma operação. Outra escolha possível, por exemplo, é disparar a regra no final de uma transação.

- Se mais de uma regra pode ser disparada pelo mesmo evento. Em caso afirmativo, a execução das regras pode ser seqüencial, apenas uma regra é executada por vez, ou concorrente.

- Se uma regra pode disparar outra regra ou (outra instância da) a mesma regra.

- Interação entre as transações geradas pelas ações das regras e execução de transações iniciadas pelo usuário. Alguns sistemas executam regras dentro do escopo da transação de disparo, isto é, da transação na qual o evento disparado ocorreu. Outros sistemas permitem um escopo mais flexível de regras e transações.

- Qual a regra que deve ser executada a partir de um conjunto de regras disparadas (*resolução de conflitos*). Normalmente a escolha é aleatória, no entanto, algumas linguagens oferecem características que podem influenciar na resolução de conflitos, tais como prioridades numéricas, hierarquia de exceção e ordenamento parcial [22, 23].

2.5 Utilizando Regras para a Manutenção de Visões Materializadas

Na literatura são encontradas muitas propostas para sistemas de bancos de dados ativos (tais como *Ariel* [24], *POSTGRES* [25], *Starburst* [26], *HIPAC* [27] e *Ode* [28]), alguns dos quais foram criados com o intuito de aperfeiçoar a manutenção incremental de visões materializadas [29, 20]. Nesses sistemas, os mecanismos ativos são altamente acoplados ao SGBD. Essa característica aumenta o desempenho do sistema porém limita a funcionalidade da regra, pois tais mecanismos ficam subordinados aos recursos específicos do SGBD.

Uma abordagem alternativa para a manutenção de visões usando regras consiste em capturar o espírito e a funcionalidade dos bancos de dados ativos sem a necessidade de estar ligado a um SGBD específico [10]. Para conseguir isso, são usados *módulos ativos* [30, 31] ou modelos de execução de regras [19, 32, 33, 34, 35, 36], os quais definem quando e como as regras disparadas devem ser executadas. Um módulo ativo é um módulo de *software* que incorpora [10]:

- uma base de regras, que especifica o comportamento do módulo de uma maneira relativamente declarativa;

- um módulo de execução, para aplicar as regras (no espírito dos bancos de dados ativos);

- um repositório de dados local, opcional.

A maioria dos sistemas de bancos de dados ativos e modelos de execução propostos foram concebidos para ambientes centralizados, em que as regras são disparadas e agem

em um mesmo banco de dados [21]. Em um ambiente de integração de dados, as regras disparadas em um banco de dados local podem agir sobre dados de outros bancos de dados. Nesse caso, mecanismos mais sofisticados para processar as regras são necessárias. Esses mecanismos devem garantir que o processamento das regras em um ambiente distribuído é equivalente ao processamento dessas regras em um ambiente centralizado correspondente [32]. Exemplos de mecanismos para ambientes distribuídos são encontrados em [21, 37, 10, 32].

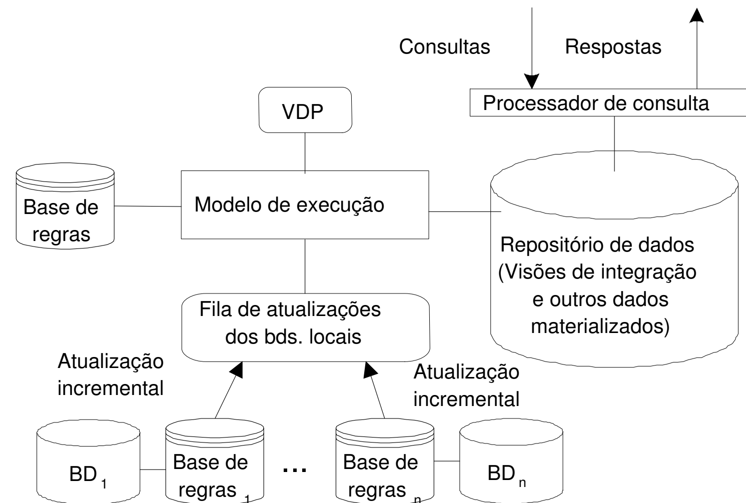

Figura 2.1: A arquitetura adotada no projeto *Squirrel* [13]

O projeto Squirrel [13] usa regras para fazer a manutenção incremental de visões de integração materializadas, as quais são gerenciadas por um modelo de execução. Nesse trabalho, existem regras para notificar as mudanças nos bancos de dados locais (as regras locais) e para atualizar a visão materializada (as regras globais). A arquitetura adotada em [13] é mostrada na Figura 2.1 e, como pode ser visto, é composta pelos seguintes componentes:

- *Bases de regras locais*: armazenam as regras locais, as quais notificam as atualizações relevantes[2] ocorridas nas bases de dados locais para o sistema global.

[2]Uma operação de atualização é relevante para uma dada classe de visão se esse tipo de atualização pode tornar a visão inconsistente.

- *Base de regras do sistema global*: armazena as regras globais geradas para o sistema global, as quais especificam as atualizações a serem realizadas na visão em conseqüência de atualizações das bases de dados locais.

- *Fila de atualização*: contém atualizações incrementais relevantes das fontes de informação remotas.

- *Plano de decomposição de visões (VDP: View Decomposition Plan)*: um VDP é um grafo acíclico direcionado, semelhante aos planos de execução usados em otimizações de consultas, que representa a decomposição de uma visão de integração suportada pelo sistema global. Cada nó do VDP representa uma classe, sendo que no topo estão as classes de visão originais, na base estão as classes das fontes de informação e nos nós intermediários estão as classes auxiliares obtidas a partir das classes de visão originais. É importante notar que as regras que especificam as atualizações a serem realizadas na visão são criadas a partir do VDP, o qual serve também como guia para executá-las em uma ordem correta.

- *Módulo de execução*: garante que as atualizações das bases de dados locais serão corretamente refletidas na visão de integração. Em essência, o modelo de execução usa o VDP como guia para a aplicação das regras do sistema global, as quais foram disparadas em decorrência das atualizações das bases de dados locais armazenadas na fila de atualização.

- *Repositório de dados*: armazena os dados da visão materializada e outros dados que possam ser úteis na manutenção da visão.

- *Processador de consultas*: é o módulo onde o usuário interage com o sistema, efetuando consultas e recebendo as respostas diretamente do repositório de dados.

A manutenção de uma visão de integração usando a arquitetura da Figura 2.1 é realizada do seguinte modo. Quando uma atualização relevante ocorre em uma base de dados local, uma regra local é disparada, cuja ação consiste em notificar a visão de integração da atualização sucedida. Essa atualização ocorrida na base de dados local é armazenada na fila de atualização. De tempos em tempos, o módulo de execução processa as atualizações na fila de atualização do seguinte modo. Para cada atualização da fila de atualização, é disparada uma regra cuja ação consiste das atualizações que devem ser feitas nas classes da visão, de modo a torná-las consistentes com a atualização ocorrida na base de dados local. É importante notar que a atualização de uma classe de visão pode tornar outras classes de visão inconsistentes (aquelas posicionadas em níveis mais elevados no VDP). Assim sendo, "novas" regras são disparadas para atualizar essas classes. O módulo de

execução coordena a execução dessas regras, usando o VDP como guia, de modo que a visão reflita a atualização ocorrida na base de dados local.

Nessa arquitetura, e como na maioria das abordagens para manutenção incremental de visões, é necessário enviar consultas às bases de dados locais a fim de obter dados adicionais para corretamente atualizar a visão. Em um ambiente distribuído, é preferível que não seja necessário enviar consultas às bases de dados locais, pois entre outros fatores, pode degradar o desempenho do sistema uma vez que envolve acesso remoto. Como mencionado anteriormente, quando uma visão pode ser mantida sem acessar as bases de dados locais, esta é denominada auto-manutenível (*self-maintainable* [14]).

Neste trabalho, será apresentada uma estratégia para a auto-manutenção, de forma incremental, de classes de fusão em visões de integração materializadas. A seguir são mostrados alguns trabalhos relacionados ao problema abordado nesta dissertação.

2.6 Trabalhos Relacionados

A maioria das abordagens para tratar o problema de manutenção incremental de visões materializadas encontradas na literatura tem focalizado ambientes centralizados [38, 12, 39, 40, 41, 42]. Em tais tipos de ambientes, um único sistema controla todas as "classes base" e "classes de visão". Assim, as atividades podem ser monitoradas de forma inteligente e o sistema pode determinar exatamente os dados necessários para atualizar a visão materializada quando ocorrerem atualizações nas fontes de informação.

Em um ambiente de integração de dados, as "classes de visão" encontram-se em um sistema separado das "classes base". Além disso, como mencionado anteriormente, podem existir fontes de informação sem atividade ou com atividade restrita. Nesse caso, não é possível determinar que dados podem ser necessários incorporar à visão materializada quando ocorrerem atualizações nas fontes de informação. Assim, normalmente consultas são enviadas as bases de dados locais, a fim de obter os dados necessários para atualizar a visão materializada. Essas consultas são avaliadas nas bases de dados locais após a atualização que a originou, de modo que as fontes de informação podem ter sido alteradas por outras atualizações nesse período. Esse desacoplamento entre as bases de dados locais e o mecanismo de manutenção da visão pode levar a visões inconsistentes [9]. Dessa forma, as estratégias focalizadas em ambientes centralizados não podem ser utilizadas em ambientes de integração de dados, entretanto, tem servido de inspiração para as novas abordagens.

A seguir são apresentados alguns trabalhos relacionados que tratam o problema da manutenção incremental de visões materializadas em ambientes centralizados (Ceri e

Widom [12], Kuno e Rudensteiner [43]) e em ambientes distribuídos (Zhou, Hull e King [13] e Gupta, Jagadish e Mumick [15]).

2.6.1 Manutenção de Visões Materializadas em Ambientes Centralizados

Em Ceri e Widom [12], é proposta uma abordagem baseada em regras para fazer a manutenção incremental de vários tipos de visões (*relações derivadas*). Essas regras são geradas usando apenas a definição das relações derivadas, isso faz com que sejam criadas regras desnecessárias. Outra característica desse enfoque é que, devido a baixa expressividade semântica do modelo relacional, é preciso fazer uma *recomputação da visão* a fim de atualizar a visão materializada. Isso significa que as atualizações a serem realizadas nas relações derivadas são calculadas usando as definições dessas relações e a porção modificada (*deltas*) das relações locais.

Ceri e Widom tratam as operações de modificação como remoções seguidas de inserções. Essa estratégia é inadequada quando múltiplos bancos de dados são utilizados, pois informações úteis para a manutenção da visão podem ser perdidas [44]. Além disso, é muito difícil reconstruir uma tupla (ou objeto) se esta é a fusão de várias tuplas de múltiplos bancos de dados, como será mostrado no próximo capítulo.

É importante salientar que no modelo relacional podem ocorrer tuplas duplicadas nas relações derivadas, devido à realização de projeções nos atributos chaves das relações locais. Para esses casos, não é possível uma manutenção incremental. Assim sendo, para tratar esse problema, Ceri e Widom geram regras para rematerializar as relações derivadas. Nas estratégias focalizadas no modelo de objetos, tal problema não ocorre pois independentemente dos valores armazenados em suas propriedades, os objetos sempre podem ser identificados por seus OIDs.

Em Kuno e Rudensteiner [43], é proposta uma abordagem baseada em algoritmos não recursivos para fazer a manutenção incremental de classes de visão orientadas a objetos. Essa abordagem visa otimizar o processo de manutenção incremental da visão materializada aproveitando os recursos inerentes ao modelo de objetos, como por exemplo a hierarquia de classes. Kuno e Rudensteiner tratam as operações de atualização tradicionais (inserção e remoção de objetos, e modificação dos valores das propriedades), além das operações de *adição e remoção de tipos de uma classe* em um objeto. Intuitivamente, "adicionar o tipo de uma classe" a um objeto significa que o objeto passa a fazer parte da extensão dessa classe. Em contrapartida, "remover o tipo de uma classe" de um objeto significa que o objeto deixa de pertencer a extensão dessa classe. Essas operações de atualização serviram de inspiração para a criação das operações "*AdicioneTipo*" e "*Re-*

movaTipo" desenvolvidas nesta dissertação, as quais são mostradas no próximo capítulo.

2.6.2 Manutenção de Visões Materializadas em Ambientes Distribuídos

Em Zhou, Hull e King [13] é proposta uma abordagem baseada em regras para fazer a manutenção incremental de visões materializadas orientadas a objetos. Essa abordagem permite que uma classe de visão tenha propriedades de classes relacionadas com suas *classes raízes*[3], porém não mencionam como os relacionamentos são tratados. Outra característica que merece destaque é que, embora esteja focalizada no paradigma orientado a objetos, toda abordagem é explicada usando apenas a álgebra relacional. Isso faz com que os recursos próprios de um modelo de objetos não sejam utilizados e muito da semântica que poderia ter sido extraída seja perdida. Assim sendo, como ocorreu em [12], é preciso fazer uma recomputação das classes de visão com base nos *deltas* das classes base, para determinar as atualizações a serem feitas na visão materializada.

Zhou, Hull e King [13] foi o único trabalho encontrado na literatura para tratar o problema de visões de integração orientadas a objetos. No entanto, sua estratégia não aborda a auto-manutenção de visões materializadas. Esse problema foi tratado apenas para visões relacionais, especificamente Gupta, Jagadish e Mumick [15] foi o único trabalho encontrado que trata do problema da auto-manutenção de *outerjoin views*.

Em Gupta, Jagadish e Mumick [15] é considerado o problema de manutenção incremental e auto-manutenção de relações derivadas do tipo *outerjoin* ("*outerjoin views*"), incluindo um tipo especial de *outerjoin* denominada "*match view*". Nesse trabalho são apresentados algoritmos procedurais para fazer a manutenção incremental de vários tipos de *outerjoin views* (*full-, left-, right-outerjoin* e *match view*). Além disso, foram identificadas as condições sob as quais uma relação derivada do tipo *full-outerjoin* é auto-manutenível. De acordo com esse trabalho uma *full-outerjoin* é auto-manutenível se satisfaz as seguintes condições: 1) contém apenas atributos próprios das relações base; 2) contém os atributos chaves das relações locais e; 3) contém os atributos que são usados em algum predicado. Já para uma *match view* ser auto-manutenível é suficiente incluir um atributo de cada relação local tal que este não pode ter valor "null" nessa relação, não pode ser um de seus atributos chaves e exista em apenas uma das relações base. Uma restrição, portanto, imposta em [15] para a auto-manutenção, é que a relação derivada só pode ter atributos de suas "relações raízes", isto é, das relações que originaram a relação derivada. Outra limitação desse enfoque é que o algoritmo proposto não poderia ser adaptado para

[3]Intuitivamente, uma classe local **A** é raiz de uma classe de visão **B** quando a extensão de **B** é obtida a partir de **A** (vide seção 3.5).

o modelo de objetos pois não teria como tratar as propriedades do tipo relacionamento. Como será mostrado no próximo capítulo, os valores de relacionamentos são objetos, e os identificadores de objetos (OIDs) locais não têm significado no sistema global.

No enfoque proposto neste trabalho são tratadas classes de fusão completas, as quais são tipos de classes de visão similares as *match view* do modelo relacional. As classes de fusão completas tratadas são mais abrangentes que as consideradas em [15], uma vez que podem ter propriedades derivadas e relacionamentos. Nesta dissertação é mostrado que classes com tais tipos de propriedades não podem ser auto-mantidas levando-se em consideração apenas as condições apresentadas em [15]. Neste trabalho é apresentada uma estratégia para tratar o problema da auto-manutenção de classes de fusão que tenham propriedades derivadas e relacionamentos.

Um resumo das características das estratégias para a manutenção incremental de visões materializadas discutidas nesta seção é apresentado na tabela 2.1.

[4] A operação de modificação é tratada como deleção seguida de inserção.

[5] Não propaga a mudança dos valores de propriedades para as classes de visão.

[6] A operação de união não é a mesma para todas as estratégias.

[7] Não é preciso criar uma classe unicamente para ocultar propriedades (fazer projeções).

[8] As *outerjoins* do presente trabalho são menos restritivas que as dos demais enfoques pois permitem a definição de propriedades derivadas e relacionamentos.

[9] O predicado de seleção pode envolver funções complexas.

Estratégias	[12]	[43]	[15]	[13]	**Enf. proposto**
Ambiente	centralizado	centralizado	distribuído	distribuído	distribuído
Modelo	relacional	OO	relacional	relacional e OO	OO
Tipo de Operações	IDM[4]	IDMAR[5]	IDM	ID	IDMAR
Regra ou algoritmo?	regra	algoritmo	algoritmo	regra	regra
Recomputação?	sim	não	sim	sim	não
Atividade do BD de origem	NA	NA	SR	SA	SA
Estratégia de manutenção	Mi e Re	Mi	Mi	Mi	Mi
Tipo de "Classes":					
SPJ,$\cup$[6],$\cap$	X	X		X	X[7]
diferença		X		X	X
outerjoin			T	Mc	T[8]
predicado de seleção que envolve outras tabelas/classes	X	X[9]			X

Notação:
I-inserção, **D**-deleção, **M**-modificação, **A**-adiciona tipo, **R**-remove tipo
SA-suficientemente ativo, **NA**-não se aplica, **SR**- sem restrições
Mi-manutenção incremental, **Re**-rematerialização
S-seleção, **P**-projeção, **J**-junção, $\cup$- união, $\cap$- interseção
T-full outerjoin, left outerjoin, right outerjoin, natural outerjoin, fusão completa
Mc- match class

Tabela 2.1: Características dos Trabalhos Relacionados e do Enfoque Proposto

Capítulo 3

Visões no Modelo de Objetos

3.1 Introdução

Este capítulo aborda visões no modelo de objetos, destacando seu uso para integração de dados. Diferentemente do que ocorre em sistemas de bancos de dados relacionais, não existe uma padronização na área de sistemas de bancos de dados orientados a objetos. Fato este, apontado por muitos como o motivo pelo qual esses sistemas não são largamente utilizados. Algumas tentativas de padronização já foram desenvolvidas, como a proposta de padronização do ODMG (*Object Data Management Group* [45]) porém, a maioria encontra-se em processo de desenvolvimento. No entanto, o modelo de objetos mostra-se muito mais atrativo que o modelo relacional para um ambiente de integração de dados pois: fornece uma maior flexibilidade e possui uma semântica mais expressiva. Além disso, segundo Papakonstantinou et al. [46], modelos e linguagens relacionais para definição de visões são insuficientes para fornecer integração de dados mesmo para bancos de dados relacionais [47].

Assim, muitos projetos tem adotado ou desenvolvido modelos de objetos para facilitar a integração de dados, independente se as bases de dados são relacionais, objeto-relacionais ou orientadas a objetos.

O capítulo é organizado como se segue:

- Na seção 3.2, é apresentado o modelo de objetos usado no presente trabalho, o qual baseia-se no modelo adotado no projeto *MultiView* [48] e no modelo de objetos do ODMG-97 [45].

- Na seção 3.3, são definidas as assertivas de correspondência, as quais especificam formalmente o relacionamento entre as visões materializadas e as classes base.

- Na seção 3.4, são apresentados os conceitos de visões orientadas a objetos e de classes de visão.

- Na seção 3.5, é mostrado como as visões podem ser definidas usando as assertivas de correspondência.

3.2 O Modelo de Objetos

Os elementos básicos de um modelo de objetos são os objetos e as classes. Os objetos representam entidades do mundo real e possuem um único identificador, chamado *Object-Identifier (OID)*. Os OIDs são gerados pelo sistema e nunca mudam mesmo quando o conteúdo do objeto é modificado. Neste trabalho, como no modelo de objetos do ODMG-97 [45], é feita uma distinção entre objetos e literais. Um literal é um tipo especial de objeto que não possui um identificador e apenas pode ser acessado via outros objetos.

Um objeto tem estado e comportamento. O estado de um objeto é definido pelos valores que o objeto possui para um conjunto de propriedades. O comportamento de um objeto é definido por um conjunto de operações (métodos) que podem ser executadas nos objetos.

Os objetos com propriedades e operações comuns são agrupados em classes. Classes servem como moldes para as suas instâncias (os objetos). Uma definição de classe possui o nome da classe (único em todo banco de dados), um tipo e uma extensão. Esta última consiste de um conjunto de objetos que são membros de uma classe. O tipo de uma classe **C** (*Tipo*(**C**)) é uma tupla <*propriedades*, *operações*> que indica o conjunto de propriedades e operações que são aplicadas a todos os objetos de uma classe. O conjunto de propriedades definidas para uma classe **C** é referenciado por *props*(**C**). A extensão de uma classe **C** é o conjunto das instâncias de **C** em um dado instante. Existem dois tipos de classes: *classes de objeto*, cujas instâncias são objetos com identificadores, e *classes de literais*, cujas instâncias são literais.

As propriedades podem ser definidas como relacionamentos funcionais entre objetos de uma classe e outros objetos (ou literais) que são instâncias da classe definida como domínio da propriedade. Por exemplo, a propriedade **depto** da classe EMPREGADO relaciona objetos da classe EMPREGADO com objetos da classe DEPARTAMENTO, logo a propriedade **depto** tem como domínio a classe DEPARTAMENTO (*Dom*(**depto**) = DEPARTAMENTO). De acordo com o tipo de classe do domínio, as propriedades podem ser classificadas em: *atributos* e *relacionamentos*. O domínio de um atributo é definido em uma classe literal. Por exemplo, o atributo **nomeEmpregado**, cujo domínio é a classe STRING, é uma propriedade que relaciona um objeto da classe EMPREGADO com uma instância da classe

STRING, que é uma classe literal. O domínio de um relacionamento é definido em uma classe de objeto. Esse é o caso da propriedade **depto** da classe EMPREGADO. No modelo de objetos do ODMG-97, um relacionamento é definido implicitamente pela declaração de *caminhos de travessia* que possibilitam as aplicações usarem conectores lógicos entre os objetos participando do relacionamento. Nesse modelo, apenas são tratados relacionamentos binários e para cada direção do relacionamento é declarado um caminho de travessia. Por exemplo, entre as classes EMPREGADO e DEPARTAMENTO existe um relacionamento representado pelas propriedades **depto**, definida em EMPREGADO e cujo domínio é DEPARTAMENTO, e **emps**, definido em DEPARTAMENTO cujo domínio é EMPREGADO. No presente trabalho, uma propriedade do tipo relacionamento representa um dos caminhos de travessia de um relacionamento no ODMG-97.

As propriedades podem ainda ser classificadas em monovaloradas ou multivaloradas. Em se tratando de uma propriedade monovalorada, cada objeto está relacionado com no máximo um objeto. O valor dessa propriedade é um objeto atômico que é uma instância do domínio da propriedade. Por exemplo, a propriedade **nomeEmpregado** da classe EMPREGADO, definida por **nomeEmpregado**: STRING, é uma propriedade monovalorada. No caso de uma propriedade multivalorada, um objeto pode estar associado a mais de uma instância do domínio da propriedade. Assim sendo, o valor de uma propriedade multivalorada é uma coleção de objetos os quais são instâncias do domínio da propriedade. Neste trabalho, como no modelo de objetos do ODMG-97, uma coleção de objetos é composta por elementos distintos, cada um dos quais pode ser um objeto atômico, outra coleção ou um literal, desde que todos os elementos da coleção sejam do mesmo tipo. Dentre os tipos de coleções tratadas no ODMG-97, pode-se citar: **array**, **set** e **list**. "*Set*" é uma estrutura de dados que captura a noção usual de conjuntos matemáticos. Os construtores "*list*" e "*array*" são estruturas que captam a noção usual de lista e vetores (tipos abstratos de dados usados em programação). Por exemplo, a propriedade **telefone** da classe EMPREGADO, definida por **telefone**: *set*<STRING>, associa um conjunto de instâncias da classe STRING a um objeto da classe EMPREGADO, indicando que um empregado pode ter mais de um número de telefone.

O esquema orientado a objetos é um conjunto de definição de classes que servem como moldes para gerar os objetos que estão envolvidos no domínio da aplicação. Na notação gráfica da Figura 3.1, retângulos representam classes. Os atributos são escritos dentro do retângulo com seus respectivos domínios. Seta simples representa relacionamento monovalorado e seta dupla representa relacionamento multivalorado, com o nome da propriedade escrito próximo a seta da classe que o define. Maiores detalhes da notação gráfica utilizada serão apresentados à medida que novos conceitos forem sendo introduzidos.

Os objetos também podem estar relacionados através de caminhos compostos por mais

de uma propriedade. Suponha, por exemplo, o esquema da Figura 3.1. Nesse esquema, tem-se que um empregado está relacionado com o gerente do seu departamento através do caminho **depto • gerente**. A seguir, é apresentada uma definição formal de caminho.

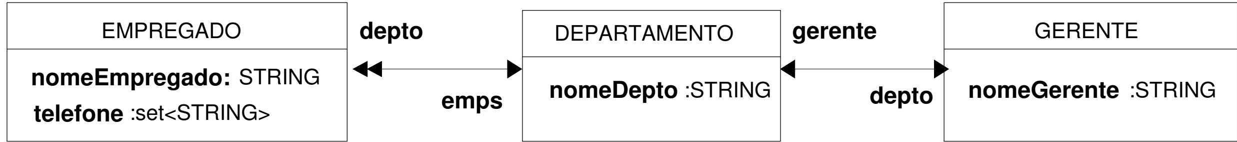

Figura 3.1: Exemplo de um Esquema OO

Definição 3.1 (*Caminho) Sejam* $\mathbf{C}_1$, $\mathbf{C}_2$, ..., $\mathbf{C}_n+1$ *classes de um esquema e* $\boldsymbol{p}_1$, ..., $\boldsymbol{p}_n$ *propriedades tais que* $\boldsymbol{p}_i \in$ *props*($\mathbf{C}_i$) *e Dom*($\boldsymbol{p}_i$) $= \mathbf{C}_{i+1}$, $1 \leq i \leq n$. *Assim sendo,* $\boldsymbol{p}_1$• $\boldsymbol{p}_2$• ... • $\boldsymbol{p}_n$ *é um caminho de* $\mathbf{C}_1$. *Isso significa que as instâncias de* $\mathbf{C}_1$ *estão relacionadas com as instâncias de* $\mathbf{C}_{i+1}$ *através do caminho* $\boldsymbol{p}_1$• $\boldsymbol{p}_2$• ... • $\boldsymbol{p}_n$.

Um caminho pode ser *monovalorado* ou *multivalorado*. Um caminho é monovalorado quando todas as propriedades que compõem o caminho são monovaloradas. Um caminho é multivalorado se uma ou mais das propriedades as quais compõem o caminho são multivaloradas. Além disso, pode-se distinguir duas categorias de caminhos: *de valor* e *de referência*. Seja o caminho ϱ=$\boldsymbol{p}_1$• $\boldsymbol{p}_2$• ... • $\boldsymbol{p}_n$. O caminho ϱ é dito de valor quando o domínio de $\boldsymbol{p}_n$ é definido em uma classe literal. O caminho ϱ é dito de referência quando o domínio de $\boldsymbol{p}_n$ é definido em uma classe de objeto.

A seguir são mostrados os conceitos de esquema e estado de esquema.

Definição 3.2 (*Esquema de Objeto) Um esquema de objetos é uma tripla* $\mathsf{S} = (\mathcal{C}, \mathcal{P}, \mathcal{I})$, *onde* $\mathcal{C}$ *representa um conjunto finito de classes,* $\mathcal{P}$ *simboliza um conjunto finito de propriedades das classes em* $\mathcal{C}$ *e* $\mathcal{I}$ *é um conjunto de restrições de integridade.*

No restante desta seção considere o esquema $\mathsf{S} = (\mathcal{C}, \mathcal{P}, \mathcal{I})$ onde **A**, **B**, $\mathbf{C}_1$, $\mathbf{C}_2$, ..., $\mathbf{C}_{n+1}$, $\mathbf{C'}_1$, $\mathbf{C'}_2$, ..., $\mathbf{C'}_{n+1}$ são classes de $\mathcal{S}$.

Definição 3.3 (*Estado de um esquema) Um estado (ou instância)* $\mathcal{D}$ *do esquema* S, *em um dado instante, é uma função definida como se segue:*

- *Para qualquer classe* $\mathbf{C}$, $\mathcal{D}(\mathbf{C})$ *(ler-se extensão da classe* $\mathbf{C}$ *no estado* $\mathcal{D}$*) é o conjunto de objetos que são instâncias de* $\mathbf{C}$ *no estado* $\mathcal{D}$.

- *Para qualquer propriedade monovalorada* $\boldsymbol{p}$ *da classe* $\mathbf{A}$ *onde Dom(*$\boldsymbol{p}$*)=*$\mathbf{B}$*,* $\mathcal{D}(\boldsymbol{p})$ *atribui uma instância* $\mathbf{b}$ *de* $\mathbf{B}$ *para cada instância* $\mathbf{a}$ *de* $\mathbf{A}$*. A notação* $\mathbf{a}.\mathcal{D}(\boldsymbol{p})$ *indica o valor da propriedade* $\boldsymbol{p}$ *para o objeto* $\mathbf{a}$ *no estado* $\mathcal{D}$*. Dado que o valor de* $\mathbf{a}.\mathcal{D}(\boldsymbol{p}) = \mathbf{b}$*, diz-se que* $\mathbf{a}$ *está relacionado com* $\mathbf{b}$ *através da propriedade* $\boldsymbol{p}$*.*

- *Para qualquer propriedade multivalorada* $\boldsymbol{p}$ *da classe* $\mathbf{A}$ *onde Dom(*$\boldsymbol{p}$*)=* $\mathbf{B}$*,* $\mathcal{D}(\boldsymbol{p})$ *atribui para cada instância* $\mathbf{a}$ *de* $\mathbf{A}$ *uma coleção de objetos* $\mathbf{c}$ *cujos membros são instâncias de* $\mathbf{B}$*. A notação* $\mathbf{a}.\mathcal{D}(\boldsymbol{p})$ *indica o valor da propriedade* $\boldsymbol{p}$ *para o objeto* $\mathbf{a}$ *no estado* $\mathcal{D}$*. Dado que o objeto* $\mathbf{b} \in \mathbf{a}.\mathcal{D}(\boldsymbol{p})$*, diz-se que* $\mathbf{a}$ *está relacionado com* $\mathbf{b}$ *através da propriedade* $\boldsymbol{p}$*.*

- *Para qualquer caminho monovalorado* $\varrho = \boldsymbol{p}_1 \bullet \boldsymbol{p}_2 \bullet ... \bullet \boldsymbol{p}_n$*, em que* $\boldsymbol{p}_i \in props(\mathbf{C}_i)$*, Dom(*$\boldsymbol{p}_i$*)=*$\mathbf{C}_{i+1}$*,* $1 \leq i \leq n$*,* $\mathcal{D}(\varrho)$ *atribui para cada objeto* $\mathbf{c}_1 \in \mathcal{D}(\mathbf{C}_1)$ *um objeto* $\mathbf{c}_{n+1} \in \mathcal{D}(\mathbf{C}_{n+1})$*, tal que existem os objetos* $\mathbf{c}_2$*, ...,* $\mathbf{c}_n$*, em que* $\mathbf{c}_i \in \mathcal{D}(\mathbf{C}_i)$*, com* $2 \leq i \leq n$*, e* $\mathbf{c}_i$ *está relacionado com* $\mathbf{c}_{i+1}$ *através da propriedade* $\boldsymbol{p}_i$*. A notação* $\mathbf{c}.\mathcal{D}(\varrho)$ *indica o valor do caminho* ϱ *para o objeto* $\mathbf{c}$ *no estado* $\mathcal{D}$*.*

- *Para qualquer caminho multivalorado* $\varrho = \boldsymbol{p}_1 \bullet \boldsymbol{p}_2 \bullet ... \bullet \boldsymbol{p}_n$*, em que* $\boldsymbol{p}_i \in rops(\mathbf{C}_i)$*, Dom(*$\boldsymbol{p}$*)=*$\mathbf{C}_{i+1}$*,* $1 \leq i \leq n$*,* $\mathcal{D}(\varrho)$ *atribui para cada objeto* $\mathbf{c}_1 \in \mathcal{D}(\mathbf{C}_1)$ *uma coleção de objetos* $\mathbf{c}_{n+1} \in \mathcal{D}(\mathbf{C}_{n+1})$*, tais que existem os objetos* $\mathbf{c}_2$*, ...,* $\mathbf{c}_n$*, em que* $\mathbf{c}_i$ *está relacionado com* $\mathbf{c}_{i+1}$ *através da propriedade* $\boldsymbol{p}_i$ *e* $\mathbf{c}_i \in \mathcal{D}(\mathbf{C}_i)$*, para* $1 \leq i \leq n$*. A notação* $\mathbf{c}.\mathcal{D}(\varrho)$ *indica o valor do caminho* ϱ *para o objeto* $\mathbf{c}$ *no estado* $\mathcal{D}$*.*

- *Para qualquer classe* $\mathbf{C}$ *de* S *e predicado* p*,* $\mathcal{D}(\mathbf{C}[\mathsf{p}])$ *é o conjunto de instâncias* $\mathbf{c}$*, tais que* $\mathbf{c} \in \mathcal{D}(\mathbf{C})$ *e* $\mathsf{p}(\mathbf{C}) = true$ *(i.e. satisfaz o predicado* p*).*

Para simplificar, as referências ao estado corrente $\mathcal{D}$ serão omitidas quando possível. Por exemplo, ao invés de $\mathbf{c}.\mathcal{D}(\varrho)$, será usado $\mathbf{c}.\varrho$. A seguir é apresentado o conceito de chave de uma classe.

Definição 3.4 (*Chave de uma classe) Um conjunto mínimo de caminhos de uma classe* $\mathbf{C}$*, que identificam um único objeto de* $\mathbf{C}$*, é denominado uma chave de* $\mathbf{C}$*. Se* $\{\varrho_1, ..., \varrho_n\}$ *é uma chave de* $\mathbf{C}$*, então para quaisquer objetos* $\mathbf{c}_1$ *e* $\mathbf{c}_2$ *de* $\mathbf{C}$*, se* $\mathbf{c}_1.\varrho_i = \mathbf{c}_2.\varrho_i$*,* $1 \leq i \leq n$*, então* $\mathbf{c}_1 \equiv \mathbf{c}_2$*, ou seja,* $\mathbf{c}_1$ *e* $\mathbf{c}_2$ *são semanticamente equivalentes (representam o mesmo objeto do mundo real). Como uma classe não pode ter objetos duplicados então* $\mathbf{c}_1 \equiv \mathbf{c}_2$ *significa que* $\mathbf{c}_1$ *e* $\mathbf{c}_2$ *são um único objeto. Isso significa que os valores da chave identificam unicamente os objetos da classe. Note que essa definição de chave permite que a chave de uma classe possa conter propriedades próprias de* $\mathbf{C}$ *(caminho com apenas uma propriedade), assim como propriedades de outras classes que estão relacionadas a* $\mathbf{C}$ *através de um caminho.*

A seguir é apresentada a definição formal dos vários tipos de assertivas de correspondência, as quais são tipos especiais de restrições de integridade que serão usadas para especificar o relacionamento entre as classes da visão de integração[1] e as classes dos esquemas base.

3.3 Assertivas de Correspondência

As Assertivas de Correspondência são tipos especiais de restrições de integridade que são usadas para estabelecer a correspondência entre os componentes de esquemas. Dessa forma, assertivas de correspondência são utilizadas para especificar que a semântica de algumas partes de um esquema está de alguma forma relacionada com a semântica de algumas partes de outro(s) esquema(s) [6].

Existem vários tipos de assertivas de correspondência, dependendo dos elementos envolvidos e da natureza da correspondência, isto é, da restrição imposta. Essas assertivas expressam o conhecimento "este elemento está relacionado com este outro elemento", e portanto impõem restrições nos estados permissíveis do banco de dados. Os estados permissíveis do banco de dados são aqueles que satisfazem a estrutura e as restrições do esquema do banco de dados.

As assertivas de correspondência foram definidas primeiramente em [6] para o modelo ER. No presente trabalho, são consideradas as assertivas de correspondência dos seguintes grupos:

* Assertivas de Correspondência de Extensão
* Assertivas de Correspondência de Propriedades
* Assertivas de Correspondência de Caminhos

No restante desta seção, considere o esquema $\mathsf{S} = (\mathcal{C}, \mathcal{P}, \mathcal{I})$, onde $\mathbf{C}$, $\mathbf{C}_1$, ... , $\mathbf{C}_{n+1}$,$\mathbf{C'}_1$, ..., $\mathbf{C'}_m$, $\mathbf{C'}_{m+1}$ são classes de S e $\mathcal{D}$ é um estado de S.

3.3.1 Assertivas de Correspondência de Extensão

As Assertivas de Correspondência de Extensão (ACE) especificam os relacionamentos existentes entre as extensões das classes de um ou mais esquemas de objetos. Existem nove tipos de ACEs as quais são definidas na tabela 3.1.

[1] Como mencionado no capítulo 1, o termo visão de integração é utilizado neste trabalho para designar um esquema de banco de dados, cujos elementos (por exemplo, relações ou classes) são obtidos a partir da integração de elementos de um ou mais esquemas de bancos de dados já especificados.

Assertivas de Correspondência de Extensão		
Relação	**Notação**	**Condição de validação em $\mathcal{D}$**
Subconjunto	$\mathbf{C}_1 \subset \mathbf{C}_2$	(i) $\mathcal{D}(\mathbf{C}_1) \subset \mathcal{D}(\mathbf{C}_2)$ (ii) Existe uma função injetiva $f : \mathcal{D}(\mathbf{C}_1) \rightarrow \mathcal{D}(\mathbf{C}_2)$
Equivalência	$\mathbf{C}_1 \equiv \mathbf{C}_2$	(i) $\mathcal{D}(\mathbf{C}_1) = \mathcal{D}(\mathbf{C}_2)$ (ii) Existe uma função bijetiva $f : \mathcal{D}(\mathbf{C}_1) \rightarrow \mathcal{D}(\mathbf{C}_2)$
Disjunção	$\mathbf{C}_1 \vert \mathbf{C}_2$	(i) $\mathcal{D}(\mathbf{C}_1) \cap \mathcal{D}(\mathbf{C}_2) = \emptyset$
Diferença	$\mathbf{C} \equiv \mathbf{C}_1$ - $\mathbf{C}_2$	(i) $\mathcal{D}(\mathbf{C}) = \mathcal{D}(\mathbf{C}_1) - \mathcal{D}(\mathbf{C}_2)$ (ii) Existe uma função bijetiva $f : \mathcal{D}(C) \rightarrow \mathcal{D}(\mathbf{C}_1) - \mathcal{D}(\mathbf{C}_2)$
União	$\mathbf{C} \equiv \cup_{i=1}^{n} \mathbf{C}_i$	(i) $\mathcal{D}(\mathbf{C}) = \cup_{i=1}^{n} \mathcal{D}(\mathbf{C}_i)$ (ii) Existe uma função bijetiva $f : \mathcal{D}(C) \rightarrow \cup_{i=1}^{n} \mathcal{D}(\mathbf{C}_i)$
Interseção	$\mathbf{C} \equiv \cap_{i=1}^{n} \mathbf{C}_i$	(i) $\mathcal{D}(\mathbf{C}) = \cap_{i=1}^{n} \mathcal{D}(\mathbf{C}_i)$ (ii) Existe uma função bijetiva $f : \mathcal{D}(C) \rightarrow \cap_{i=1}^{n} \mathcal{D}(\mathbf{C}_i)$
Seleção	$\mathbf{C}_1 \equiv \mathbf{C}_2[\mathsf{p}]$	(i) $\mathcal{D}(\mathbf{C}_1) = \mathcal{D}(\mathbf{C}_2[\mathsf{p}])$ (ii) Existe uma função bijetiva $f : \mathcal{D}(\mathbf{C}_1) \rightarrow \mathcal{D}(\mathbf{C}_2[\mathsf{p}])$
Sobreposição	$\mathbf{C}_1 \oslash \mathbf{C}_2$	(i) $\mathcal{D}(\mathbf{C}_1) \cap \mathcal{D}(\mathbf{C}_2) \neq \emptyset$ ($\mathbf{C}_1$ e $\mathbf{C}_2$ podem ter instâncias em comum) (ii) Existe uma função injetiva parcial $f : \mathcal{D}(\mathbf{C}_1) \rightarrow \mathcal{D}(\mathbf{C}_2)$
Generalização	$\mathbf{C} \equiv Gen(\mathbf{C}_1, ..., \mathbf{C}_n)$	(i) $C = \cup_{i=1}^{n} \mathbf{C}_i (C \equiv \cup_{i=1}^{n} \mathbf{C}_i)$ (ii) $C = \mathbf{C}_i \cap \mathbf{C}_j = \emptyset$, para $i \neq j (\mathbf{C}_i \mid \mathbf{C}_j)$

Tabela 3.1: Assertivas de Correspondência de Extensão

Na tabela 3.1, para cada tipo de ACE são definidas a notação utilizada para sua especificação e as condições que devem ser satisfeitas para sua validação em um dado estado $\mathcal{D}$. A letra "p" aparecendo na assertiva de correspondência de seleção representa o predicado de seleção que deve ser satisfeito para o objeto ser membro de uma classe da visão de integração.

As ACEs particionam as classes dos esquemas em classes de equivalência. Duas classes $\mathbf{C}_1$ e $\mathbf{C}_2$ são "*semanticamente equivalente*" se para qualquer estado $\mathcal{D}$ existir uma função de mapeamento 1-1 ($\mathsf{F} : \mathcal{D}(\mathbf{C}_1) \rightarrow \mathcal{D}(\mathbf{C}_2)$) entre os objetos de $\mathbf{C}_1$ e $\mathbf{C}_2$. Pode existir outras funções de mapeamento, possivelmente relacionando múltiplos objetos de uma classe a um objeto de outra classe. Em geral, $\mathbf{C}_1$ e $\mathbf{C}_2$ possuem um identificador comum, o qual é usado como a função de mapeamento. Se um objeto $\mathbf{c}_1$ de $\mathbf{C}_1$ é mapeado para um objeto $\mathbf{c}_2$ de $\mathbf{C}_2$ ($\mathcal{F} : (\mathbf{c}_1) = \mathbf{c}_2$), então $\mathbf{c}_1$ e $\mathbf{c}_2$ são "*semanticamente equivalente*" ($\mathbf{c}_1 \equiv \mathbf{c}_2$), isto é, $\mathbf{c}_1$ e $\mathbf{c}_2$ representam a mesma entidade do mundo real. A identificação de objetos em bancos de dados distintos normalmente é baseada em comparações entre atributos [37]. Mais a respeito de funções de mapeamento e identificação de objetos pode ser encontrado na seção 3.4.

3.3.2 Assertivas de Correspondência de Propriedades

As Assertivas de Correspondência de Propriedade (ACP) (e de caminho) especificam os relacionamentos entre os tipos das classes de um ou mais esquemas.

Propriedades Semanticamente Equivalentes

Definição 3.5 (*Propriedades com Domínios Compatíveis) Seja* $\boldsymbol{p}_1 \in props(\mathbf{C}_1)$ *e* $\boldsymbol{p}_2 \in props(\mathbf{C}_2)$ *em que* $\mathbf{C}_1$ *e* $\mathbf{C}_2$ *são semanticamente equivalentes. A restrição de equivalência* $\mathbf{C}_1.\boldsymbol{p}_1 \equiv \mathbf{C}_2.\boldsymbol{p}_2$ *é válida em um estado* $\mathcal{D}$ *se e somente se para qualquer* $\mathbf{c}_1$ *e* $\mathbf{c}_2$ *tal que* $\mathbf{c}_1 \in \mathcal{D}(\mathbf{C}_1)$ *e* $\mathbf{c}_2 \in \mathcal{D}(\mathbf{C}_2)$*, se* $\mathbf{c}_1 \equiv \mathbf{c}_2$ *então* $\mathbf{c}_1.\boldsymbol{p}_1 = \mathbf{c}_2.\boldsymbol{p}_2$*.*

Quando a propriedade é um atributo, deve-se também considerar os atributos com domínios diferentes. Nesse caso, existe uma função de mapeamento de domínio (γ), tal que a restrição de equivalência $\mathbf{C}_1.\boldsymbol{p}_1 \equiv \mathbf{C}_2.\boldsymbol{p}_2$ é válida em um estado $\mathcal{D}$ se e somente se para qualquer $\mathbf{c}_1$ e $\mathbf{c}_2$ tal que $\mathbf{c}_1 \in \mathcal{D}(\mathbf{C}_1)$ e $\mathbf{c}_2 \in \mathcal{D}(\mathbf{C}_2)$, se $\mathbf{c}_1 \equiv \mathbf{c}_2$ então $\mathbf{c}_1.\boldsymbol{p}_1 = \mathbf{c}_2.(\boldsymbol{p}_2 \circ \gamma)$.

3.3.3 Assertivas de Correspondência de Caminhos

Durante a análise dos inter-relacionamentos dos esquemas também deve-se considerar a equivalência semântica dos caminhos. É importante notar que as ACPs são um caso particular de Assertivas de Correspondência de Caminho (ACCs), quando os caminhos são compostos de uma única propriedade.

Definição 3.6 (*A.C. de caminho) Considere o caminho* $\varrho_1 = \boldsymbol{p}_1 \bullet ... \bullet p_n$ *onde* $p_i \in props(\mathbf{C}_i)$ *e* $Dom(p_i) = C_{i+1}$*, e o caminho* $\varrho_2 = p'_1 \bullet ... \bullet p'_m$ *em que* $p'_i \in props(C'_i)$ *e* $Dom(p'_i) = C'_{i+1}$*, para* $1 \leq i \leq max(n, m)$*, de modo que* $\mathbf{C}_1$ *e* C'_1 *são "semanticamente equivalentes", assim como* C_{n+1} *e* C_{m+1}*. A restrição de equivalência* $\mathbf{C}_1.\varrho_1 \equiv \mathbf{C'}_1.\varrho_2$ *é válida em um estado* $\mathcal{D}$ *se e somente se para qualquer* $\mathbf{c}_1$ *e* $\mathbf{c'}_1$ *tal que* $\mathbf{c}_1 \in \mathcal{D}(\mathbf{C}_1)$ *e* $\mathbf{c'}_1 \in \mathcal{D}(C'_1)$*, se* $\mathbf{c}_1 \equiv \mathbf{c'}_1$ *então* $\mathbf{c}_1.\varrho_1 = \mathbf{c'}_1.\varrho_2$*.*

A seguir, é mostrado um exemplo do uso das ACCs.

Exemplo 3.1 *Suponha os esquemas da Figura 3.2. A ACC* ESTUDANTE$_2$.**depto$_2$** $\equiv$ ESTUDANTE$_1$.**curso$_1$** •**depto$_1$** • **nomeDepto$_1$** *especifica que o nome do departamento de um estudante no esquema* S_2 *é equivalente ao nome do departamento desse estudante no esquema* S_1*. Já a ACC* ESTUDANTE$_1$.**curso$_1$** • **depto$_1$** • **nomeDepto$_1$** $\equiv$ ESTUDANTE$_3$.**depto$_3$** • **nmDepto$_3$** *especifica que o nome do departamento de um estudante no esquema* S_1 *é equivalente ao nome do departamento desse estudante no esquema* S_3*.*

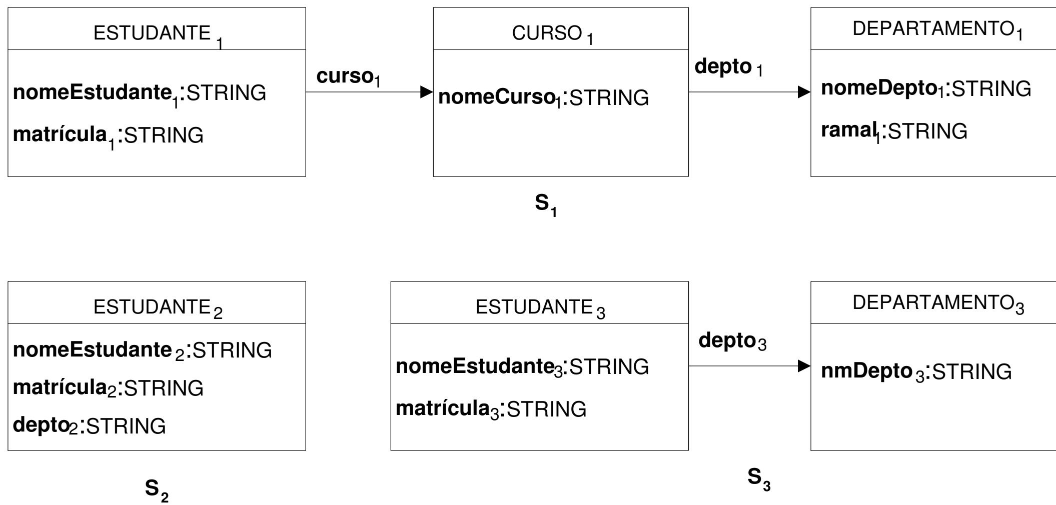

Figura 3.2: Caminhos "semanticamente equivalentes"

Uma ACC da forma $\mathbf{C}_1.\boldsymbol{p} \equiv \mathbf{C}_2.\boldsymbol{p}_1 \bullet ... \bullet \boldsymbol{p}_n$ especifica o caminho de derivação da propriedade derivada $\boldsymbol{p}$. Uma propriedade é chamada de derivada se ela for semanticamente equivalente a um caminho contendo mais de uma propriedade. No esquema da Figura 3.2, $\mathbf{depto}_2$ é uma propriedade derivada e seu caminho de derivação é mostrado no exemplo 3.1.

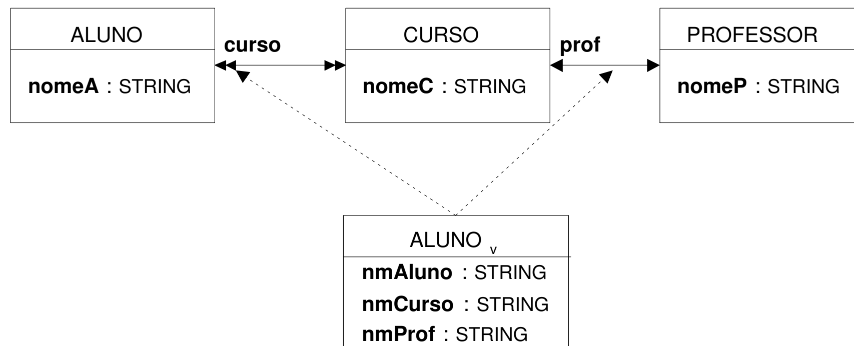

Figura 3.3: Exemplo de classes que usam operações de junção

É importante notar que o valor do atributo derivado no modelo de objetos não é descoberto através de operações de junção entre as classes, como acontece no modelo relacional. Isso ocorre porque no modelo de objetos um relacionamento entre classes é especificado explicitamente através de uma propriedade do tipo relacionamento. As operações de junções são necessárias apenas quando não há uma correspondência de um

para um entre os objetos das classes consideradas, como pode ser visto na figura 3.3. A classe ALUNO$_v$ na Figura 3.3 é formada pela junção das classes ALUNO e PROFESSOR. Como pode ser observado, um mesmo aluno pode ter mais de um curso. Assim sendo, podem existir mais de um objeto na classe ALUNO$_v$ representando um mesmo objeto de ALUNO.

3.4 Visões de Integração de Dados e Classes de Fusão

De um modo geral, visões são interfaces através das quais os usuários podem consultar (e algumas vezes atualizar) um ou mais bancos de dados. Visões são amplamente utilizadas pois, entre outros fatores, podem representar partes menores de um grande esquema, tornam transparente o acesso a dados de múltiplos bancos de dados e permitem modificações dinâmicas de um esquema de banco de dados sem perder as versões antigas. Em sistemas de bancos de dados múltiplos, visões (ou mediadores) tem sido utilizadas para integrar dados distribuídos em diferentes bancos de dados heterogêneos.

No modelo de objetos, um esquema de visão consiste de um conjunto de definições de classes de visão [48]. A definição de uma classe de visão é idêntica a das demais classes de um modelo de objetos, como definido na seção 3.2.

Como mencionado anteriormente, uma visão pode ser implementada como virtual ou materializada. No caso de visões virtuais, as extensões das classes de visão são geradas somente quando consultas são submetidas à visão. Essas extensões são obtidas a partir do mapeador de estados da visão (definição da visão), que especifica como o estado dos bancos de dados locais em um determinado instante é mapeado no estado da visão correspondente. Em ambientes onde existe um único banco de dados, uma linguagem de consulta (SQL, OQL, ...) é usada para a definição da visão, ou seja para a definição do mapeador de estados. Em ambientes de múltiplos bancos de dados, o mapeador de estados (definição da visão) é especificado de forma declarativa através de linguagens de alto nível. Esse é o caso, por exemplo, do sistema de mediação MedMaker [46] e do projeto Squirrel [13], que utilizam as respectivas linguagens MSL (*Mediator Specification Language*) e ISL (*Integration Specification Language*). No sistema MedMaker, o mapeador de estados consiste de um conjunto de regras que definem como os objetos dos tipos do mediador são obtidos a partir dos objetos dos seus tipos base. No projeto Squirrel, a definição da visão é feita de modo semelhante ao usado em sistemas centralizados, pois a ISL é baseada na ODL e OQL do padrão ODMG-97. No caso de visões materializadas, como mencionado anteriormente (capítulo 2), as extensões das classes de visão são armazenadas em um repositório de dados e devem ser mantidas em resposta as atualizações nos bancos de dados locais, a fim de ficarem consistentes com as fontes de informação.

Um tipo de classe que ocorre com freqüência em visões de integração são as chamadas *classes de fusão*, as quais correspondem a um tipo especial de "*outerjoin view*", que são definidas por um "*natural outer-equijoin nos atributos chaves*". Em uma classe de fusão, os dados referentes a um objeto dessa classe podem estar armazenados de forma distribuída, possivelmente com informações replicadas, em vários bancos de dados locais. Considere, por exemplo, os bancos de dados locais com os esquemas $\mathbf{S}_1$ e $\mathbf{S}_2$ mostrados na Figura 3.4 e o esquema $\mathbf{S}_v$ da visão **V** mostrados na Figura 3.5. A classe de visão EST&EMP é uma classe de fusão cujos objetos são obtidos da fusão dos objetos de ESTUDANTE e EMPREGADO, da seguinte forma: no caso de um estudante também ser um empregado, então existirá um único objeto correspondente na classe de visão EST&EMP cujos valores para as propriedades **nome**, **telefone**, **curso** e **rg** são obtidos do objeto correspondente em ESTUDANTE e os valores das propriedades **salário**, **dependentes** e **nomeDepto** são obtidos do objeto correspondente em EMPREGADO. No caso de um estudante não ser um empregado, então existirá um objeto correspondente na classe de visão EST&EMP cujas propriedades **salário**, **dependentes** e **nomeDepto** terão valores "nulos". No caso de um empregado não ser um estudante, então existirá um objeto correspondente na classe de visão EST&EMP cujas propriedades **telefone** e **curso** terão valores "nulos". À operação de sintetizar informações de dois ou mais objetos de diferentes bancos de dados e que representam a mesma entidade do mundo real em um único objeto da visão é denominada *fusão de objetos* (*object fusion*) [49].

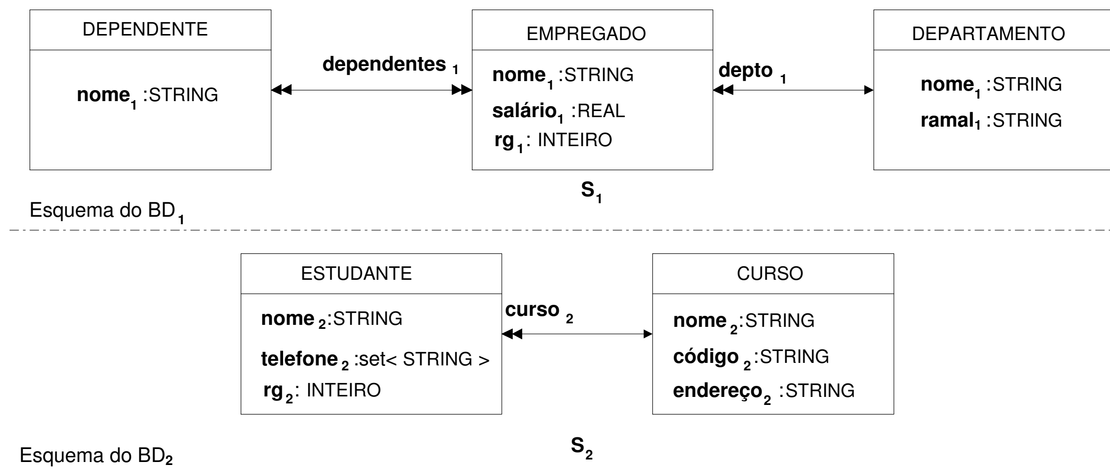

Figura 3.4: Esquemas locais $\mathbf{S}_1$ e $\mathbf{S}_2$

A fusão de objetos torna-se árdua quando as origens não são estruturadas ou são semi-estruturadas e não se tem o conhecimento de seus conteúdos e estruturas. Além disso, quando se lida com bases de dados autônomas e heterogêneas, pode não ser fácil identificar

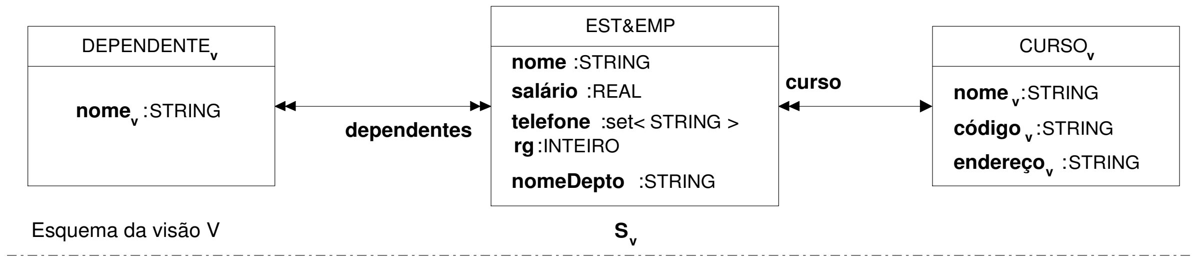

Figura 3.5: Esquema S_v da visão **V**

os objetos que são semanticamente equivalentes. Nesse caso, pode ser preciso usar funções complexas que de algum modo combinam as informações necessárias em um objeto. Fusões de objetos mais simples são possíveis quando chaves estão disponíveis nos objetos das classes base, mesmo se essas chaves estiverem representadas diferentemente nos bancos de dados locais. Por exemplo, na Figura 3.4 um objeto $\mathbf{e}_1$ da classe EMPREGADO e um objeto $\mathbf{e}_2$ da classe ESTUDANTE são semanticamente equivalentes ($\mathbf{e}_1 \equiv \mathbf{e}_2$) quando eles tem o mesmo valor para seus atributos chaves ($\mathbf{e}_1.\mathbf{rg}_1 = \mathbf{e}_2.\mathbf{rg}_2$). Quando as propriedades estiverem representadas de forma diferente, os valores das chaves podem ser mapeados para uma forma canônica, de modo que possam ser usados para formar a identidade semântica.

O "*matching*" de objetos (*object matching* [49]), ou seja, a identificação de objetos em bancos de dados distintos que representam o mesmo objeto do mundo real, é um passo fundamental no processo de fusão de objetos. Na literatura são encontradas diversas estratégias para mapear objetos em bancos de dados distintos e relacionar aqueles que são semanticamente equivalentes, algumas das quais são descritas a seguir.

- *Baseada em chaves*: essa é a estratégia mais trivial, pois confia na igualdade das chaves de dois objetos, possivelmente envolvendo atributos de outras classes (atributos derivados), para determinar a equivalência semântica dos objetos das bases de dados. É o caso, por exemplo, dos atributos chaves $\mathbf{rg}_1$ de EMPREGADO e $\mathbf{rg}_2$ de ESTUDANTE. Essa técnica é utilizada em [50, 51, 3, 10].

- *Baseada em tabelas lookups*: essas tabelas contém pares de OIDs ou atributos chaves de objetos semanticamente equivalentes. Os trabalhos apresentados em [50, 52, 10] usam essas tabelas.

- *Baseada em comparação*: comparações lógicas ou aritméticas, bem como funções definidas pelo usuário que tomam atributos como argumentos e retornam valores booleanos, são usadas para determinar a equivalência semântica entre os objetos.

É o caso, por exemplo, de uma operação que recebe como argumento os nomes de pessoas e datas de nascimento e retorna um valor booleano. Os trabalhos [52, 10, 53] suportam essa técnica.

- *Baseada em informações históricas*: equivalências semânticas anteriores entre os objetos podem ser utilizadas em adição a outros tipos de função de *matching*. Por exemplo, uma aplicação pode especificar que dois objetos já identificados como semanticamente equivalentes podem continuar assim mesmo se deixam de satisfazer outras condições de equivalência semântica. O trabalho [10] suporta essa técnica.

O "*matching*" de objetos é um agravante no uso de visões virtuais para a integração de dados. Isso porque na abordagem de visões virtuais, a identificação da correspondência semântica entre os objetos deve ser realizada a cada consulta efetuada sobre a visão. Por outro lado, na abordagem de visões materializadas, uma vez identificada a equivalência semântica entre os objetos, eles são integrados e armazenados fisicamente, podendo ser consultados quando preciso. Dessa forma, o tempo de resposta às consultas feitas sobre a visão é minimizado. Entretanto, o problema com essas classes de fusão é que, em geral, é necessário enviar consultas as bases de dados locais a fim de obter dados adicionais para corretamente atualizar a visão, como será mostrado no próximo capítulo.

3.4.1 Definindo Classes de Fusão

No modelo relacional existe o operador *outerjoin* para definir "*outerjoin views*" [15]. No modelo de objetos, tal operador não foi ainda definido para a OQL (ou suas extensões). No entanto, as classes de fusão podem ser especificadas através de regras, como as suportadas na linguagem MSL [46].

Na Figura 3.6 é apresentada a especificação em MSL da classe de visão EST&EMP (Figura 3.5), a qual consiste de cinco regras. Cada regra consiste de uma "cabeça" e de um "corpo" separados pelo símbolo :–. A cabeça da regra diz como construir os objetos da classe de visão EST&EMP. A regra $\mathbf{R}_1$ especifica como criar (virtualmente) instâncias de EST&EMP através da fusão dos objetos de ESTUDANTE e EMPREGADO. De acordo com a regra $\mathbf{R}_1$, tem-se: existe um objeto **e** em EST&EMP, onde **e.nome** = N, **e.salário** = S, **e.telefone** = T, **e.rg** = R, **e.curso** = C, **e.dependentes** = D, **e.nomeDepto** = ND, se existir um objeto $\mathbf{e}_1$ em EMPREGADO tal que $\mathbf{e}_1.\mathbf{nome}_1$ = N, $\mathbf{e}_1.\mathbf{salário}_1$ = S, $\mathbf{e}_1.\mathbf{rg}_1$ = R, $\mathbf{e}_1.\mathbf{dependentes}_1$ = D, $\mathbf{e}_1.(\mathbf{depto}_1 \bullet \mathbf{nome}_1)$ = ND e existir um objeto $\mathbf{e}_2$ em ESTUDANTE tal que $\mathbf{e}_2.\mathbf{nome}_2$ = N, $\mathbf{e}_2.\mathbf{telefone}_2$ = T, $\mathbf{e}_2.\mathbf{rg}_2$ = R, $\mathbf{e}_2.\mathbf{curso}_2$ = C.

A regra $\mathbf{R}_2$ armazena na relação RG_ESTUDANTE o número do registro geral de todos os estudantes. Essa relação serve como resultado intermediário para ser usado pela regra $\mathbf{R}_3$.

R$_1$:

<EST&EMP{<**nome** N> <**salario** S > <**telefone** T> <**rg** R> <**curso** C>
<**dependentes** D> <**nomeDepto** ND>}>

:– <EMPREGADO{<**nome**$_1$ N> <**salario**$_1$ S> <**rg**$_1$ R> <**dependentes**$_1$ D>
<**depto**$_1$ • **nome**$_1$ ND>}>

and <ESTUDANTE{<**nome**$_2$ N> <**telefone**$_2$ T> <**rg**$_2$ R> <**curso**$_2$ C>}>

R$_2$:

RG_ESTUDANTE(R) :– <ESTUDANTE {<**rg**$_2$ R>}>

R$_3$:

<EST&EMP{<**nome** N> <**salario** S > <**telefone** ‘null’> <**rg** R> <**curso** ‘null’>
<**dependentes** D> <**nomeDepto** ND>}>

:– <EMPREGADO{<**nome**$_1$ N> <**salario**$_1$ S> <**rg**$_1$ R> <**dependentes**$_1$ D>
<**depto**$_1$ • **nome**$_1$ ND>}>

and not RG_ESTUDANTE(R)

R$_4$:

RG_EMPREGADO(R) :– <EMPREGADO {<**rg**$_1$ R>}>

R$_5$:

<EST&EMP{<**nome** N> <**salario** ‘null’ > <**telefone** T> <**rg** R> <**curso** C>
<**dependentes** ‘null’> <**nomeDepto** ‘null’>}>

:– <ESTUDANTE{<**nome**$_2$ N> <**telefone**$_2$ T> <**rg**$_2$ R> <**curso**$_2$ C>}>

and not RG_EMPREGADO(R)

Figura 3.6: Especificação da classe de visão EST&EMP na linguagem MSL [46]

A regra $\mathbf{R}_3$ especifica como criar objetos de EST&EMP obtidos exclusivamente de EMPREGADO. De acordo com a regra $\mathbf{R}_3$, tem-se: existe um objeto **e** em EST&EMP, onde **e**.**nome** = N, **e**.**salário** = S, **e**.**telefone** = 'null', **e**.**rg** = R, **e**.**curso** = 'null', **e**.**dependentes** = D, **e**.**nomeDepto** = ND, se existir um objeto $\mathbf{e}_1$ em EMPREGADO tal que $\mathbf{e}_1.\mathbf{nome}_1$ = N, $\mathbf{e}_1.\mathbf{salário}_1$ = S, $\mathbf{e}_1.\mathbf{rg}_1$ = R, $\mathbf{e}_1.\mathbf{dependentes}_1$ = D, $\mathbf{e}_1.(\mathbf{depto}_1 \bullet \mathbf{nome}_1)$ = ND e não existir um objeto correspondente a $\mathbf{e}_1$ em ESTUDANTE. Note que a relação RG_ESTUDANTE, usada para verificar a existência do objeto $\mathbf{e}_1$ em EMPREGADO, impede que exista em EST&EMP dois objetos representando o mesmo objeto do mundo real (um obtido pela regra $\mathbf{R}_1$ e o outro pela regra $\mathbf{R}_3$).

A regra $\mathbf{R}_4$ armazena na relação RG_EMPREGADO o número do registro geral de todos os empregados. Essa relação serve como resultado intermediário para ser usado pela regra $\mathbf{R}_5$.

A regra $\mathbf{R}_5$ especifica como criar objetos de EST&EMP obtidos exclusivamente de ESTUDANTE. De acordo com a regra $\mathbf{R}_5$, tem-se: existe um objeto **e** em EST&EMP, onde **e**.**nome** = N, **e**.**salário** = 'null', **e**.**telefone** = T, **e**.**rg** = R, **e**.**curso** = C, **e**.**dependentes** = 'null', **e**.**nomeDepto** = 'null', se existir um objeto $\mathbf{e}_1$ em ESTUDANTE tal que $\mathbf{e}_1.\mathbf{nome}_2$ = N, $\mathbf{e}_1.\mathbf{telefone}_2$ = T, $\mathbf{e}_1.\mathbf{rg}_2$ = R, $\mathbf{e}_1.\mathbf{curso}_2$ = C e não existir um objeto correspondente a $\mathbf{e}_1$ em EMPREGADO. Semelhante ao ocorrido na regra $\mathbf{R}_3$, a relação RG_EMPREGADO, usada para verificar a existência do objeto $\mathbf{e}_1$ em ESTUDANTE, impede que exista em EST&EMP dois objetos representando o mesmo objeto do mundo real (um obtido pela regra $\mathbf{R}_1$ e o outro pela regra $\mathbf{R}_5$).

No presente trabalho, as classes de fusão não precisam ser definidas, pois estas classes são materializadas. Apenas devem ser geradas as regras para manter as classes de fusão. Essas regras, como será mostrado no próximo capítulo, são definidas com base nas assertivas de correspondência (ACs), as quais especificam formalmente o relacionamento entre as classes de visão e as classes base.

3.5 Usando as Assertivas de Correspondência para Definir Classes de Visão

Alguns enfoques utilizam uma álgebra de objetos para especificar como as classes de visão estão relacionadas com as classes base. No enfoque proposto, são usadas as assertivas de correspondência, as quais especificam formalmente o relacionamento entre as classes de visão e as classes base.

Como mostrado na seção 3.3, as assertivas de correspondência podem ser: de extensão (ACE), de propriedades (ACP) e de caminhos (ACC). As ACEs são usadas para especificar

quais os objetos das classes base serão membros da visão. As *classes raízes* de uma classe de visão são as que estão relacionadas com a classe de visão através de alguma ACE, o que significa que nas classes raízes existem objetos que são semanticamente equivalentes a objetos da classe de visão. De acordo com o tipo de relacionamento entre uma classe de visão e suas classes raízes, são distinguidos seis tipos diferentes de classes de visão [54]: *equivalência*, *seleção*, *união*, *diferença*, *interseção* e *sobreposição* (veja seção 3.3). É importante notar que para uma dada classe de visão pode existir mais de uma ACE especificada. Considere, por exemplo, os esquemas S_1 e S_2 mostrados na Figura 3.4 e os esquemas das visões $\mathbf{V}_1$ e $\mathbf{V}_2$ mostrados na Figura 3.7. A classe de visão ALUNO$_v$ na Figura 3.7 contém informações sobre estudantes que estão armazenadas nos objetos das classes ESTUDANTE e EMPREGADO. O relacionamento entre essas classes é especificado pelas ACEs Ψ_1:ALUNO$_v$ $\equiv$ ESTUDANTE e Ψ_2:ALUNO$_v$ $\oslash$ EMPREGADO. Ψ_1 especifica que ALUNO$_v$ e ESTUDANTE são equivalentes; isto é, que para cada objeto de ESTUDANTE existe um objeto semanticamente equivalente em ALUNO$_v$. Ψ_2 especifica que as classes ALUNO$_v$ e EMPREGADO podem ter objetos em comum.

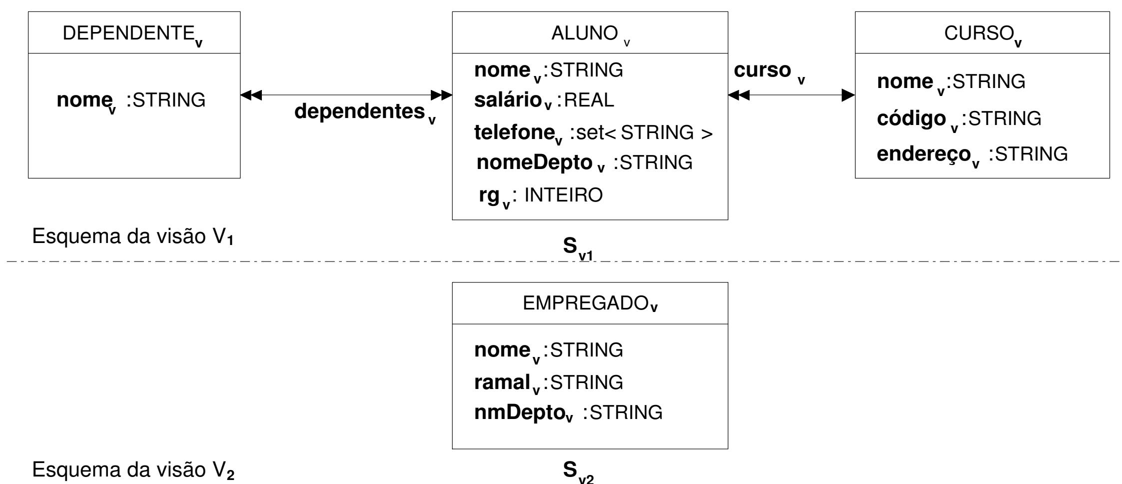

Figura 3.7: Esquema da visão S_v

As ACPs e ACCs especificam como os valores das propriedades dos objetos da classe de visão são derivados a partir dos valores das propriedades dos objetos das classes base. Por exemplo, a propriedade **salário**$_v$ da classe de visão ALUNO$_v$ (Figura 3.7) é definida pela ACP ALUNO$_v$.**salário**$_v$$\equiv$EMPREGADO.**salário**$_1$, a qual especifica que dada uma instância **a** de ALUNO$_v$, se existir uma instância **e** em EMPREGADO tal que **a**$\equiv$**e** então **a**.**salário**$_v$=**e**.**salário**$_1$. Caso contrário, **a**.**salário**$_v$= *null*. A propriedade **nomeDepto**$_v$ da classe ALUNO$_v$ é definida pela ACC ALUNO$_v$.**nomeDepto**$_v$$\equiv$EMPREGADO.**depto**$_1$$\bullet$**nome**$_1$, a qual especifica que dada uma instância **a** de ALUNO$_v$, se existir uma instância **e**

em EMPREGADO tal que **a**≡**e** então **a.nomeDepto**$_v$ = **e.depto**$_1$.**nome**$_1$. Caso contrário, **a.nomeDepto**$_v$= *null*. É importante notar que pode existir mais de uma ACP e/ou ACC especificada para uma mesma propriedade da classe de visão. Por exemplo, a propriedade **nome**$_v$ da classe de visão ALUNO$_v$ possui as seguintes ACPs: ALUNO$_v$.**nome**$_v$ ≡ EMPREGADO.**nome**$_1$ e ALUNO$_v$.**nome**$_v$≡ESTUDANTE.**nome**$_2$. Isso significa que **nome**$_1$, **nome**$_2$ e **nome**$_v$ são propriedades sinônimas e assim sendo devem ter o mesmo valor. Note que uma dada propriedade da classe de visão só terá valor definido se essa propriedade for definida em uma das classes raízes que o objeto é membro, caso contrário a propriedade terá valor "*null*".

Em uma classe de visão pode-se distinguir três tipos de propriedades, a saber:

- ***Herdada***: uma propriedade de uma classe de visão é uma propriedade "*herdada*" se esta for semanticamente equivalente a uma propriedade de uma de suas classes raízes, isto é, as propriedades estão relacionadas através das ACPs.

- ***Computada***: uma propriedade de uma classe de visão é uma propriedade computada quando tem associada a ela algumas funções ou expressões aritméticas. Esse tipo de propriedade não é tratado no presente trabalho.

- ***De junção ou derivada***: uma propriedade de uma classe de visão é uma propriedade de "*junção*" ou "*derivada*" se ela for semanticamente equivalente a um caminho (composto de mais de uma propriedade) de uma de suas classes raízes, isto é, as propriedades estão relacionadas através de uma ACC. No presente trabalho, apenas são tratadas propriedades semanticamente equivalentes a caminhos monovalorados.

  Uma classe de visão é chamada de "*classe de junção*" se ela tem uma ou mais propriedades de junção. Isso significa que uma classe de junção possui propriedades que não são propriedades dos objetos de suas classes raízes, mas de objetos relacionados com os objetos das suas classes raízes. Na Figura 3.7, a classe de visão ALUNO$_v$ é uma classe de junção, visto que possui a propriedade **nomeDepto**$_v$ cujo valor é obtido de objetos da classe DEPARTAMENTO relacionados aos objetos da classe raiz EMPREGADO.

Capítulo 4

Auto-Manutenção de Classes de Fusão Completas

4.1 Introdução

Um dos problema com a manutenção de classes de fusão é que na maioria dos casos elas não são auto-manuteníveis. É o caso da classe de fusão ALUNO$_v$ da visão $\mathbf{S}_{v_2}$ na Figura 3.7. A regra global para manutenção de ALUNO$_v$ com relação as inserções em ESTUDANTE é mostrada na Figura 4.1. Segundo esta regra, a inserção de um objeto $\mathbf{e}_1$ em ESTUDANTE acarretará em uma inserção na classe de visão ALUNO$_v$. Porém, antes de realizar a inserção do objeto correspondente em ALUNO$_v$, deve-se primeiro verificar se já existe um objeto $\mathbf{e}_2$ em EMPREGADO tal que $\mathbf{e}_1$ "*match*" $\mathbf{e}_2$ ($\mathbf{e}_1 \equiv \mathbf{e}_2$). Caso exista, insere-se em ALUNO$_v$ o objeto resultante da fusão de $\mathbf{e}_1$ com $\mathbf{e}_2$. Caso contrário, insere-se em ALUNO$_v$ o objeto criado a partir de $\mathbf{e}_1$. O "*matching*" de dois objetos [49], na maioria dos casos é realizado usando os valores dos atributos de uma chave comum, porém, [13, 49] propõem o uso de heurísticas e funções aritméticas ou booleanas para determinar a equivalência semântica entre objetos.

Quando inserção($\mathbf{e}_1$) em ESTUDANTE /* $\mathbf{e}_1$ é o objeto inserido em ESTUDANTE */

Então Se existe $\mathbf{e}_2$ em EMPREGADO tal que $\mathbf{e}_1 \equiv \mathbf{e}_2$ então
  Insira um objeto $\mathbf{a}$ em ALUNO$_v$ tal que $\mathbf{a}$ é a fusão de $\mathbf{e}_1$ e $\mathbf{e}_2$
 Senão
  Insira um objeto $\mathbf{a}$ em ALUNO$_v$ tal que $\mathbf{a} \equiv \mathbf{e}_1$

Figura 4.1: Regra para manutenção da classe de visão ALUNO$_v$ nas inserções em ESTUDANTE

Neste capítulo é apresentado o enfoque proposto para permitir a auto-manutenção de classes de fusão completas, as quais são um tipo especial de classes de fusão bastante utilizadas em integração de dados. O capítulo é organizado como se segue:

- Na seção 4.2 é definido o conceito de classes de fusão completas.
- Na seção 4.3, é apresentada a arquitetura proposta para permitir a auto-manutenção de classes de fusão completas.
- Na seção 4.4 são mostradas as regras que devem ser geradas para fazer a manutenção das classes de fusão completa.

4.2 Classes de Fusão Completas

Uma classe de fusão completa tem propriedades similares a de uma visão relacional definida por um "*natural full outer-equijoin nos atributos chaves*", denominada "*match view*" em [15] e denominada "match class" em [13]. Uma classe de fusão **F** é completa quando sua extensão é definida pela união das extensões de suas classes raízes. Mais formalmente, tem-se que:

- A extensão de **F** é definida pela assertiva de correspondência de extensão $\mathbf{F} \equiv \mathbf{C}_1 \cup ... \cup \mathbf{C}_m$, onde $\mathbf{C}_1, ..., \mathbf{C}_m$ são as classes raízes de **F**. Note que, como uma classe não pode ter objetos duplicados, então se $\mathbf{c}_i$ é uma instância de $\mathbf{C}_i$ e $\mathbf{c}_j$ é uma instância de $\mathbf{C}_j$ tal que $\mathbf{c}_i \equiv \mathbf{c}_j$, para $1 \leq$ i, j $\leq$ m e i $\neq$ j, então existirá um único objeto **f** na extensão de **F** tal que $\mathbf{f} \equiv \mathbf{c}_i \equiv \mathbf{c}_j$.
- As propriedades de **F** podem ser herdadas ou derivadas, consistindo em um subconjunto das propriedades (atributos ou relacionamentos) de suas classes raízes $\mathbf{C}_1$, ..., $\mathbf{C}_m$. Assim sendo, as propriedades de **F** são especificadas através das assertivas de correspondência de propriedades e de caminhos que relacionam as propriedades de **F** com as propriedades de suas classes raízes.

A classe EST&EMP mostrada na Figura 3.5 é um exemplo de uma classe de fusão completa, enquanto que a classe ALUNO_v (Figura 3.7) não é completa. Na Figura 4.2, é mostrada a classe EST&EMP com as assertivas de correspondência entre EST&EMP e suas classes raízes (ESTUDANTE e EMPREGADO, mostradas na Figura 3.4).

Em [15] são definidas condições em que uma "*match view*" é auto-manutenível. Traduzindo essas condições do modelo relacional para o modelo de objetos, tem-se que:

1. A classe de fusão completa **F** deve possuir os atributos das classes raízes que são necessários para determinar a equivalência semântica entre os objetos.

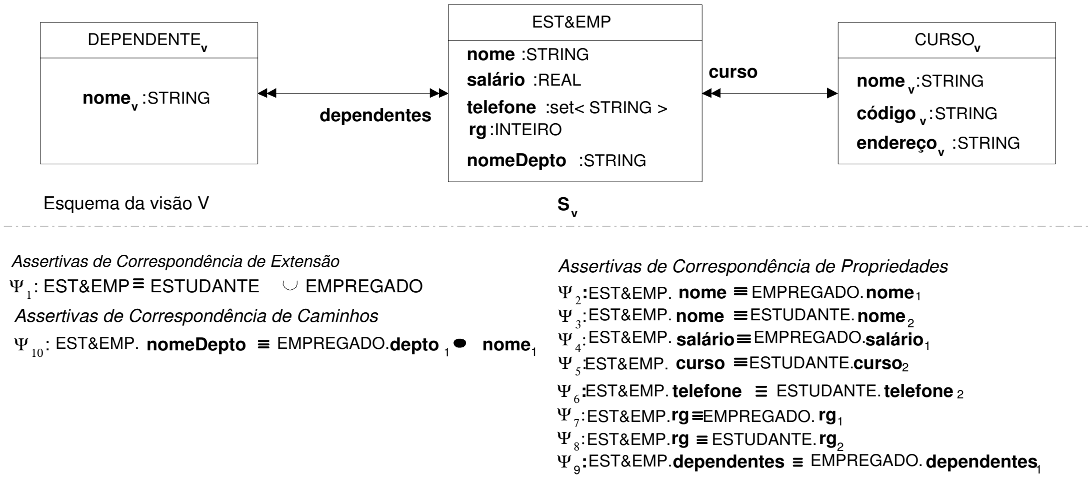

Figura 4.2: Esquema da visão S_v com as ACs entre EST&EMP e suas classes raízes

2. **F** deve ter pelo menos um atributo de cada classe raiz, tal que esse atributo não tenha valor nulo nessa classe e não seja um atributo chave (vide definição 3.4).

3. **F** pode ter somente atributos herdados.

Essas condições, por terem sido originalmente definidas para o modelo relacional, não tratam das propriedades do tipo relacionamento. Além disso, devido a restrição 3, as classes de fusão completas também não podem ter propriedades derivadas. Caso as classes de fusão completas possuam tais tipos de propriedades derivadas e relacionamentos, então estas não podem ser auto-mantidas. Considere, por exemplo, os esquemas das Figuras 3.4 e 4.2 e a inserção de um objeto $\mathbf{e}_1$ em EMPREGADO. Essa operação de atualização acarretará em uma inserção na classe EST&EMP. Para fazer a manutenção de EST&EMP, é preciso verificar se já existe um objeto $\mathbf{e}_2$ em EST&EMP tal que $\mathbf{e}_2$ "*match*" $\mathbf{e}_1$ ($\mathbf{e}_2 \equiv \mathbf{e}_1$). Isso é feito comparando as propriedades **rg** de EST&EMP e $\mathbf{rg}_1$ de EMPREGADO, por isso a restrição 1 é necessária. Caso tal objeto $\mathbf{e}_2$ exista, então os objetos $\mathbf{e}_1$ e $\mathbf{e}_2$ são "sintetizados", constituindo um único objeto em EST&EMP. Caso contrário, um novo objeto é criado em EST&EMP a partir de $\mathbf{e}_1$. Em qualquer um dos casos, são atribuídos valores para as propriedades **nome**, **salário**, **rg**, **dependentes** e **nomeDepto** da classe EST&EMP.

Os valores dos atributos herdados **nome**, **salário** e **rg** são literais e portanto obtidos diretamente das propriedades correspondentes de EMPREGADO. Os valores das propriedades do tipo relacionamento são objetos, cujas referências não têm qualquer significado no sistema global. Assim sendo, para obter o valor do relacionamento **dependentes** é necessário consultar a classe base DEPENDENTES a fim de obter as informações

necessárias para fazer o "*match*" do objeto dessa classe com o objeto da classe de visão correspondente. Os valores das propriedades derivadas são obtidos através de expressões de caminhos. Nesse caso também são necessárias consultas às fontes de informação. Assim, o valor da propriedade derivada **nomeDepto** é obtido através da expressão $\mathbf{depto}_1$ • $\mathbf{nome}_1$, a qual retorna o nome do departamento armazenado na classe DEPARTAMENTO.

Note que se apenas os atributos próprios de ESTUDANTE e EMPREGADO fossem definidos em EST&EMP, então os valores das propriedades do objeto inserido em EMPREGADO seriam usados diretamente para atualizar a classe EST&EMP e nenhum acesso às fontes de informação seria necessário.

Neste trabalho, essas restrições são resolvidas e é proposta uma estratégia que permite a auto-manutenção de classes de fusão completas que possuem propriedades (atributos e relacionamentos) herdadas e derivadas.

4.3 Enfoque Proposto para a Auto-Manutenção de Classes de Fusão Completas

Para fazer a manutenção incremental de classes de fusão completas e sem necessitar acessar os bancos de dados locais, é proposta a arquitetura mostrada na Figura 4.3.

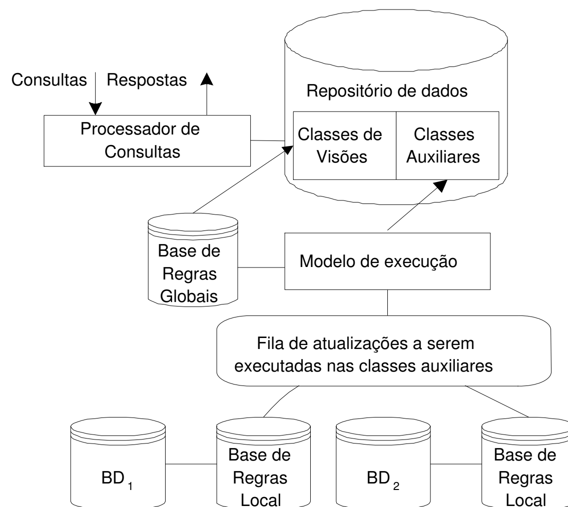

Figura 4.3: Arquitetura proposta para a auto-manutenção de classes de fusão completas

A arquitetura proposta baseia-se na arquitetura do projeto Squirrel [13] contudo, a abordagem desta dissertação diferencia-se da proposta em [13] nos seguintes aspectos:

- As bases de regras locais armazenam as regras locais para a manutenção das classes de fusão completas. Essas regras devem determinar o efeito que as atualizações locais terão sob as classes de fusão completas e propagar, para o sistema global, as atualizações que devem ser realizadas na visão a fim de torná-la consistente com relação aos bancos de dados locais. Essas atualizações a serem executadas nas classes de fusão completas ficam armazenadas na fila de atualizações (FA). Isso difere da abordagem do projeto Squirrel [13] em que as regras locais apenas notificam para o sistema global as atualizações locais relevantes. Outras regras são então definidas no sistema global, as quais determinam as atualizações que devem ser feitas na visão de modo que esta reflita a atualização ocorrida no banco de dados local. Nesse enfoque, uma visão é auto-manutenível com respeito a uma atualização τ em um banco de dados local $\mathbf{D}$ quando o efeito que τ deverá ter na classe de visão pode ser determinado usando apenas as informações da visão. A vantagem do enfoque proposto com relação ao do Squirrel, como será mostrado na seção 4.4, é que, para auto-manter a visão, além das informações da visão, também pode-se utilizar as informações no banco de dados local onde ocorreu a atualização.

  No enfoque proposto, como será mostrado na seção 4.4, não é preciso nenhum acesso às fontes de informação locais pois as ações requeridas, para atualizar a visão, são determinadas pelos bancos de dados locais. Dessa forma, as informações necessárias para atualizar a visão já podem ser extraídas dos bancos de dados locais e enviadas para o sistema global.

A manutenção de uma visão de integração usando a arquitetura proposta consiste em dois passos:

1. Quando uma atualização relevante[1] τ ocorre em um banco de dados local $\mathbf{D}$ uma regra local é disparada, a qual envia para a Fila de Atualizações, FA, as atualizações que devem ser realizadas na visão a fim de torná-la consistente o novo estado de $\mathbf{D}$.

2. Periodicamente, o módulo de execução processa as atualizações da fila atualizando desse modo as classes de visão.

Neste trabalho, para garantir a consistência do sistema global, são consideradas as seguintes suposições:

- As regras locais disparadas para uma dada operação de atualização local são executadas em uma mesma transação.

[1]Uma atualização em um banco de dados local é relevante para uma classe de visão se o novo estado do banco de dados local torna-se inconsistente com relação ao estado da classe de visão.

- Os bancos de dados locais estão consistentes entre si.

- Caso existam regras de bancos de dados locais para atualizar outros bancos de dados locais e essas atualizações sejam relevantes para a visão de integração, então cada banco de dados local disparará a regra local correspondente à atualização na classe de visão afetada pela atualização local.

- Caso existam classes replicadas nos bancos de dados locais, então apenas atualizações relevantes na classe primária podem disparar as regras locais correspondentes para fazer a manutenção das classes de fusão completas.

Uma vantagem do enfoque proposto é que o processo de manutenção de visão no sistema global é realizado com maior rapidez por dois motivos principais:

- Não é preciso acessar os bancos de dados locais;

- Não é necessário calcular no sistema global que atualizações são necessárias para fazer a manutenção da visão de integração, pois essas já foram previamente computadas nos bancos de dados locais.

Um processo de manutenção mais rápido no sistema global é muito útil, por exemplo, em sistemas de suporte à decisão, pois normalmente esses sistemas não disponibilizam os dados para os usuários quando atualizações estão sendo processadas.

Contudo, a arquitetura proposta não resolve completamente o problema da auto-manutenção de classes de fusão completas para todos os tipos de atualizações locais. Existem situações, como será mostrado na próxima seção, onde para manter corretamente a visão, é necessário conhecer de quais classes raízes um determinado objeto da visão é membro. Infelizmente, essa informação não pode ser obtida diretamente da classe de fusão completa. Para resolver esse problema é adotada a seguinte estratégia: uma nova propriedade, denominada **classes**, é inserida na classe de fusão completa, a qual tem como valor o conjunto de nomes das classes raízes das quais um objeto é membro (**classes**:set<STRING>). Assim sendo, uma classe de fusão completa **F**, além da ACE $\mathbf{F} \equiv \mathbf{C}_1 \cup ... \cup \mathbf{C}_m$, tem as seguintes ACEs de seleção: $\mathbf{C}_k \equiv \mathbf{F}[\text{“}\mathbf{C}_k\text{”} \in \mathbf{classes}]$, $1 \leq k \leq m$, as quais especificam que uma instância **f** de **F** é membro da classe raiz $\mathbf{C}_k$ se e somente se “$\mathbf{C}_k$” $\in$ **f.classes**.

Como será mostrado mais adiante, classes de fusão completas com a propriedade **classes** são auto-manuteníveis com relação a qualquer atualização τ em um banco de dados local **D**, pois o efeito que τ deverá ter na classe de visão pode ser determinado utilizando apenas as informações de **D** e da visão. Classes de fusão que não são completas não são auto-manuteníveis. Considere, por exemplo, a classe de fusão ALUNO_v na Figura

3.7. Note que a regra da Figura 4.1 requer um acesso a $\mathbf{BD}_1$ para verificar se já existe um objeto em EMPREGADO tal que esse objeto seja semanticamente equivalente ao objeto inserido em ESTUDANTE. Essa informação não pode ser obtida localmente (em $\mathbf{BD}_2$) nem na visão. Logo ALUNO_v não é auto-manutenível. No entanto, como será mostrado no próximo capítulo, ALUNO_v pode tornar-se auto-manutenível quando a classe de fusão completa EST&EMP for usada como classe auxiliar.

Na próxima seção, são apresentadas as regras locais que devem ser geradas para a manutenção de classes de fusão completas. Como poderá ser observado, essas regras determinam localmente todas as informações necessárias para fazer a manutenção da classe de fusão completa, e mantém a visão utilizando essas informações e as informações na visão. Porém, antes de mostrar as regras, é preciso definir o conceito de *conjunto de propriedades que se aplicam a um objeto de uma classe de fusão completa.*

Como foi mostrado na seção 3.4, um objeto de uma classe de fusão **F** nem sempre tem valor definido para todas suas propriedades. Na realidade, um objeto de uma classe de fusão completa **F** terá valor definido para uma propriedade ***p*** de **F** se esta propriedade for definida em uma das classes raízes que o objeto é membro. Nesse caso, diz-se que a propriedade ***p*** se aplica ao objeto. Formalmente tem-se que uma propriedade ***p*** de **F** se aplica a um objeto **f** de **F** se e somente se existe "**C**" $\in$ **f.classes** e existe uma propriedade ***p'*** de **C** tal que **F**.***p*** $\equiv$ **C**.***p'*** ou **F**.***p*** $\equiv$ **C**.***p'*** $\bullet\ \varrho$, em que ϱ é um caminho. Então, tem-se que o conjunto de propriedades que se aplicam ao objeto **f** de uma classe de fusão completa **F** consiste do conjunto de propriedades de **F** que se aplicam a **f**. Caso uma propriedade ***p*** não se aplique ao objeto **f**, então ***p*** terá valor nulo.

4.4 Gerando as Regras para a Manutenção das Classes de Fusão Completas

No presente trabalho, as regras para fazer a manutenção das classes de visão são definidas a partir das assertivas de correspondência, as quais especificam formalmente o relacionamento entre o esquema da visão e os esquemas locais. A seguir, será mostrado como gerar regras de produção a partir das assertivas de correspondência. Na seção 4.4.2, são definidas quais as regras que devem ser geradas para a manutenção de uma classe de fusão completa.

4.4.1 Gerando Regras para a Manutenção Incremental de Visões a partir das Assertivas de Correspondência

Uma questão fundamental no uso de regras para a manutenção incremental de visões materializadas é a geração das regras. O processo de geração de regras consiste nos seguintes passos:

1. Determinar quais as operações de atualização nos bancos de dados locais que podem tornar a visão inconsistente, isto é, as operações relevantes.
2. Para cada operação de atualização relevante nos bancos de dados locais, deverá ser gerada uma regra, cujo evento consiste da referida operação e cuja ação consiste em enviar ao sistema global as atualizações que devem ser executadas na visão a fim de que a visão reflita a atualização ocorrida no banco de dados local.

Alguns enfoques usam a definição da visão, isto é, o mapeador de estados, para gerar as regras que farão sua manutenção [12, 13]. Neste trabalho, as regras são definidas a partir das assertivas de correspondência da visão, as quais especificam formalmente o relacionamento entre as classes de visão e as classes base. No enfoque proposto, as assertivas de correspondência são tratadas como tipos especiais de restrições de integridade que devem ser preservadas pelas operações de atualização das fontes de informação locais. Assim, dada uma operação de atualização em um banco de dados local, é preciso definir a seqüência de atualizações na visão necessárias para torná-la "sincronizada" com o novo estado dos bancos de dados locais. Esse problema consiste em determinar a seqüência de atualizações na visão requerida para a manutenção de cada assertiva de correspondência da visão que são relevantes para essa operação. Uma assertiva de correspondência é relevante para uma operação se ela pode ser violada por essa operação.

Assim sendo, as regras necessárias para a manutenção incremental de uma dada visão são estas requeridas para a manutenção das assertivas de correspondência da visão. Assim, o problema de determinar as regras para a manutenção de uma visão fica, portanto, reduzido a determinar, para cada assertiva de correspondência da visão, o conjunto de regras necessárias para a sua manutenção.

O problema para definir as regras necessárias para a manutenção de uma dada assertiva de correspondência consiste de dois passos:

1. Primeiro, identifica-se as operações de atualização nos bancos de dados locais que podem violar a assertiva de correspondência, isto é, as operações relevantes para a assertiva de correspondência;

2. Para cada operação relevante detectada em (1), deve ser gerada uma regra, cujo evento consiste da referida operação e cuja ação consiste em enviar ao sistema global a seqüência de atualizações a serem executadas na visão de forma que a assertiva de correspondência seja preservada.

O uso das assertivas de correspondência permite provar formalmente que as regras geradas no enfoque proposto realizam corretamente a manutenção da visão. Isso significa que as atualizações requisitadas na visão pelas regras locais refletem exatamente o efeito necessário para que a visão fique consistente com as fontes de informação locais.

4.4.2 Gerando as Regras para a Manutenção das Classes de Fusão Completas a partir das Assertivas de Correspondência

Nesta seção, considere **F** uma classe de fusão completa tal que **F** herda propriedades das classes raízes $\mathbf{C}_1$, ..., $\mathbf{C}_m$ e de mais nenhuma outra e considere **M** o módulo mediador do sistema global.

Como discutido na seção 4.4.1, as regras necessárias para fazer a manutenção de uma classe de visão são estas requeridas para fazer a manutenção das assertivas de correspondência dessa classe de visão. Assim, para fazer a manutenção da classe de fusão completa **F** devem ser geradas regras para fazer a manutenção das assertivas de correspondência definidas nas seções 4.2 e 4.3, as quais especificam formalmente o relacionamento entre **F** e suas classes raízes $\mathbf{C}_1$, ..., $\mathbf{C}_m$. A seguir, são apresentadas as regras que devem ser geradas para a manutenção das Assertivas de Correspondência de Extensão (ACE), das Assertivas de Correspondência de Propriedades (ACP), e das Assertivas de Correspondência de Caminhos (ACC) de **F**.

• Manutenção das Assertivas de Correspondência de Extensão

De acordo com o exposto nas seções 4.2 e 4.3, tem-se que as ACEs de **F** são:

- Ψ_k:$\mathbf{C}_k \equiv \mathbf{F}$[“$\mathbf{C}_k$” $\in$ **classes**], $1 \leq$ k $\leq$ m, e

- σ:$\mathbf{F} \equiv \mathbf{C}_1 \cup ... \cup \mathbf{C}_m$.

A ACE σ é derivada a partir das ACEs Ψ_1, ..., Ψ_m. Logo, tem-se que a assertiva de correspondência σ é preservada quando as assertivas de correspondência Ψ_k são preservadas. Assim sendo, apenas é preciso gerar regras para a manutenção de Ψ_k, as quais são definidas a seguir.

Regras para a manutenção de Ψ_k

Como mostrado na seção 4.4.1, deve ser gerada uma regra para cada operação de atualização relevante de Ψ_k. No caso da assertiva de correspondência Ψ_k, as operações de atualização relevantes são: remoções e inserções[2] de objetos em $\mathbf{C}_k$, cujas regras são mostradas a seguir.

Regra para remoções em $\mathbf{C}_k$

A Figura 4.4 mostra a regra que deve ser gerada para fazer a manutenção da assertiva de correspondência Ψ_k com relação às remoções em $\mathbf{C}_k$. Note que o evento dessa regra está parametrizado com o objeto **c** que foi removido de $\mathbf{C}_k$, de forma que esse objeto pode ser referenciado na ação da regra[3]. A ação da regra consiste primeiramente em gerar a *especificação de identificador de objeto* *id* de **F**, a qual será usada para fazer o "*matching*" do objeto **c** removido da classe raiz $\mathbf{C}_k$ com um objeto em **F** que é semanticamente equivalente a **c** (linha 2 na Figura 4.4). A estrutura de uma Especificação de Identificador de Objeto (EIO) de uma classe **F** é formada pelo subconjunto das propriedades de **F** cujos valores unicamente identificam os objetos dessa classe e são obtidos a partir dos valores das propriedades de **c**. Para gerar a EIO *id*, é chamado o método "*GereEIO_de_***F**" da classe raiz $\mathbf{C}_k$. Note que o objeto **c** removido de $\mathbf{C}_k$ não pode ser utilizado diretamente para identificar o objeto **f** em **F**, tal que $\mathbf{f} \equiv \mathbf{c}$, uma vez que a chave de $\mathbf{C}_k$ pode ser constituída por propriedades derivadas ou relacionamentos.

---

01. *Quando* remoção(**c**) de $\mathbf{C}_k$ /* **c** é o objeto removido de $\mathbf{C}_k$ */
02. *Então* $id = \mathbf{c}.GereEIO_de_\mathbf{F}()$;
03. Requisita as seguintes atualizações para **M**:
04. <Selecione **f** de **F** usando *id*; /* (**f**≡**c**) */
05. **f**.*RemovaTipo*("$\mathbf{C}_k$");>

Figura 4.4: Regra para manutenção de Ψ_k nas remoções em $\mathbf{C}_k$

Para ilustrar o processo descrito, considere a classe de fusão completa EST&EMP apresentada na figura 3.5 e suas classes raízes ESTUDANTE e EMPREGADO mostradas na figura 3.4. As assertivas de correspondência entre EST&EMP e suas classes raízes são definidas na figura 4.5. A regra gerada para fazer a manutenção da ACE Ψ_2:EMPREGADO $\equiv$ EST&EMP

[2] Para simplificar, neste trabalho, escreve-se inserções/remoções em $\mathbf{C}_k$ ao invés de inserção/remoção na extensão de $\mathbf{C}_k$.

[3] Em alguns sistemas, como o Hipac [27], os eventos de uma regra podem ser parametrizados e esses parâmetros podem ser referenciados na condição e na ação da regra. Em um sistema como o Oracle8, o objeto **c** é referenciado como "*old*".

[“EMPREGADO” ∈ **classes**] com relação às remoções em EMPREGADO é mostrada na Figura 4.6. Na Figura 4.7 é apresentado o método “*GereEIO_de_*EST&EMP” da classe EMPREGADO que gera uma EIO de EST&EMP, a qual é usada para fazer o “*matching*” do objeto **c** removido de EMPREGADO com o objeto em EST&EMP que é semanticamente equivalente a **c**. Lembre-se que as funções de “*matching*” podem ser tão simples como uma comparação entre atributos chaves. Esse é o caso da função “*GereEIO_de_*EST&EMP” onde as classes EMPREGADO e EST&EMP possuem um identificador comum (os respectivos atributos chaves $\mathbf{rg}_1$ e **rg**). Dessa forma, como mostrado na Figura 4.7, o método “*GereEIO_de_*EST&EMP” é bastante simples, bastando atribuir um valor ao atributo chave para gerar uma EIO de EST&EMP. No entanto, como discutido anteriormente (seção 3.4), as funções de “*matching*” também podem envolver heurísticas e funções aritméticas ou booleanas mais complexas.

ACs de Extensão

Ψ_1:ESTUDANTE≡EST&EMP[“ESTUDANTE”∈**classes**]

Ψ_2:EMPREGADO≡EST&EMP[“EMPREGADO”∈**classes**]

Ψ_3:EST&EMP≡ESTUDANTE ∪ EMPREGADO

ACs de Propriedades

Ψ_4:EST&EMP.**nome**≡EMPREGADO.$\mathbf{nome}_1$

Ψ_5:EST&EMP.**nome**≡ESTUDANTE.$\mathbf{nome}_2$

Ψ_6:EST&EMP.**salario**≡EMPREGADO.$\mathbf{salario}_1$

Ψ_7:EST&EMP.**curso**≡ESTUDANTE.$\mathbf{curso}_2$

ACs de Propriedades

Ψ_8:EST&EMP.**telefone**≡ESTUDANTE.$\mathbf{telefone}_2$

Ψ_9:EST&EMP.**rg**≡EMPREGADO.$\mathbf{rg}_1$

Ψ_{10}:EST&EMP.**rg**≡ESTUDANTE.$\mathbf{rg}_2$

Ψ_{11}:EST&EMP.**dependentes**≡EMPREGADO.$\mathbf{dependentes}_1$

ACs de Caminho

Ψ_{12}:EST&EMP.**nomeDepto**≡EMPREGADO.$\mathbf{depto}_1$ •$\mathbf{nome}_1$

Figura 4.5: Assertivas de Correspondência entre EST&EMP e suas classes raízes

---

Quando remoção(**c**) de EMPREGADO /* **c** é o objeto removido de EMPREGADO */

Então *id* = **c**.*GereEIO_de_*EST&EMP();

Requisita as seguintes atualizações para **M**:

<Selecione **e** de EST&EMP usando *id*; /* (**e**≡**c**) */

e.*RemovaTipo*(“EMPREGADO”);>

Figura 4.6: Regra para manutenção de Ψ_2 nas remoções em EMPREGADO

Na regra da Figura 4.4, uma vez gerada a especificação de identificador de objeto *id* de **F**, as atualizações a serem realizadas na classe **F** são então enviadas para o mediador (linhas 3-5 na Figura 4.4), as quais são descritas a seguir:

```
GereEIO_de_EST&EMP();
{
v_rg = this.rg_1;
retorne((rg: v_rg));
}
```

Figura 4.7: Método *GereEIO_de*_EST&EMP

1. selecionar o objeto **f** de **F** tal que **f** é semanticamente equivalente ao objeto **c** (**c** é o objeto que foi removido de $\mathbf{C}_k$) usando *id*.

2. chamar o método "*RemovaTipo*", onde o nome da classe "$\mathbf{C}_k$" é passado como parâmetro. O método "*RemovaTipo*", apresentado na Figura 4.8, terá os seguintes efeitos no objeto **f**:

   (a) Remove **C** do conjunto de classes de **f** (**f.classes**)[4]. Lembre-se que **C** é o nome da classe da qual o objeto foi removido.

   (b) Caso **f** não seja membro de mais nenhuma classe das bases de dados locais (**f.classes** $==\emptyset$) então **f** é removido de **F**; caso contrário, o objeto **f** "perde o tipo" da classe **C**, isto é, valores "null" são atribuídos as propriedades $\boldsymbol{p}$ que não se aplicam mais para o objeto **f** ($\boldsymbol{p} \notin$ **f**.*props*()[5]). Neste trabalho, atribuir "null" a uma propriedade significa atribuir o *valor nulo* no caso de propriedades monovaloradas ou atribuir o *conjunto vazio* no caso de propriedades multivaloradas.

Regra para inserções em $\mathbf{C}_k$

A Figura 4.9 mostra a regra que deve ser gerada para fazer a manutenção da ACE Ψ_k:$\mathbf{C}_k \equiv \mathbf{F}[$"$\mathbf{C}_k$" $\in$ **classes**$]$ com relação às inserções em $\mathbf{C}_k$. Note que o evento dessa regra está parametrizado com o objeto **c** que foi inserido em $\mathbf{C}_k$. A ação da regra consiste primeiramente em gerar a *especificação do objeto* **c'** a ser inserido na classe **F** (linha 2 na Figura 4.9)[6]. Para gerar a Especificação de Objeto (EO) **c'**, é chamado o método "*GereEO_de_***F**" da classe $\mathbf{C}_k$, o qual retorna a especificação de objeto **c'** de **F**, tal que **c'** "*match*" **c** (**c'**≡**c**). A estrutura da especificação de objeto **c'** é formada por todas as propriedades de **F** e a propriedade **ClasseDeOrigem** cujo valor é o nome da classe

[4] Para a remoção de valores em um conjunto é usado o comando *remove_element* do padrão ODMG-97.

[5] "*props*" é o método da classe de fusão completa **F** que determina o conjunto de propriedades de **F** que se aplicam a um objeto **f** de **F** (vide seção **??**).

[6] Em um sistema como o Oracle8, **c** é referenciado como "*new*"

RemovaTipo(STRING **C**);
/* O objeto "perde o tipo" de **C** ou é removido de sua classe */
{
this.**classes**.*remove_element*(**C**);
Se **this**.**classes** == $\emptyset$ então
  Remova **this** de sua classe;
Senão
  Para cada propriedade $\boldsymbol{p} \in$ **this**.**propriedades** tal que **this**.$\boldsymbol{p} \neq$ null e $\boldsymbol{p} \notin$ **this**.*props*()
faça /* **this**.**propriedades** são as props. do obj. quer se apliquem ou não a este */
    this.$\boldsymbol{p}$= null;
}

Figura 4.8: Método *RemovaTipo*

"$\mathbf{C}_k$" onde o objeto **c** foi inserido. Note que a estrutura de uma EIO é formada apenas pelas propriedades usadas para fazer o "*matching*" dos objetos, enquanto que a estrutura de uma EO é formada por todas as propriedades de **F**. Outra observação importante é que a operação "*GereEO_de_***F**" deve ser executada localmente para ser possível tratar o problema das propriedades derivadas e relacionamentos.

01. *Quando* inserção(**c**) em $\mathbf{C}_k$ /* **c** é o objeto inserido em $\mathbf{C}_k$ */
02. *Então* **c'**= **c**.*GereEO_de_***F**();
03.   Requisita as seguintes atualizações para **M**:
04.   <Se existe **f** em **F** tal que **f**≡**c'** então /* **Caso 1** */
05.   **f**.*AdicioneTipo*(**c'**);
06.   Senão /* **Caso 2** */
07.   **F**.*CrieObjeto*(**c'**);>

Figura 4.9: Regra para manutenção de Ψ_k nas inserções em $\mathbf{C}_k$

Na Figura 4.10 é mostrada a regra gerada para fazer a manutenção da ACE Ψ_2:EMPREGADO ≡ EST&EMP ["EMPREGADO" ∈ **classes**] com relação às inserções em EMPREGADO. Na Figura 4.11 é apresentado o método "*GereEO_de_*EST&EMP" da classe EMPREGADO. O método "*GereEO_de_*EST&EMP" retorna uma EO de EST&EMP, onde os valores atribuídos as propriedades de EST&EMP (**nome**, **salário**, **rg**, **telefone**, **curso**, **dependentes** e **nomeDepto**) são definidos a partir dos valores das propriedades do objeto **c** inserido em EMPREGADO. Isso é feito com base nas ACPs e ACCs que especificam

a correspondência entre as propriedades de EST&EMP e EMPREGADO. Mais adiante será discutido detalhadamente como isso é feito.

Quando inserção(**c**) em EMPREGADO   /* **c** é o objeto inserido em EMPREGADO */
Então   **c'** = **c**.*GereEO_de_*EST&EMP();
    Requisita as seguintes atualizações para **M**:
        <Se existe **e** em EST&EMP tal que **e**≡**c'** então
            e.*AdicioneTipo*(**c'**);
        Senão
            EST&EMP.*CrieObjeto*(**c'**);>

Figura 4.10: Regra para manutenção de Ψ_2 nas inserções em EMPREGADO

---

*GereEO_de_*EST&EMP();
{
01. Inicializa as variáveis correspondentes às propriedades **nome**, **salário**, **rg**, **telefone**, **curso**, **dependentes** e **nomeDepto** com valores "null"
02. v_{nome} =**this**.**nome**$_1$;
03. $v_{salario}$ =**this**.**salário**$_1$;
04. v_{rg} =**this**.**rg**$_1$;
05. Para cada objeto **v** em **this**.**dependentes**$_1$ faça
06. $\mathbf{v}_{dependentes}$.*insert_element*(**v**.*GereEIO_de_*DEPENDENTE$_v$());
07. $v_{nomeDepto}$ =**this**.**depto**$_1$.**nome**$_1$;
08. retorne((**nome**: v_{nome}, **salário**: $v_{salario}$, **rg**: v_{rg}, **telefone**: $v_{telefone}$, **curso**: v_{curso}, **dependentes**: $v_{dependentes}$, **nomeDepto**: $v_{nomeDepto}$, **ClasseDeOrigem**: "EMPREGADO"));
}

Figura 4.11: Método *GereEO_de_*EST&EMP

Na regra da Figura 4.9, uma vez gerada a especificação de objeto **c'** de **F**, as atualizações a serem realizadas na classe **F** são então enviadas para o mediador (linhas 3-7 na Figura 4.9), as quais consistem em um dos casos descritos abaixo.

Caso 1: *Existe um objeto em* **F** *semanticamente equivalente a* **c'**

No caso de já existir um objeto **f** em **F**, tal que **f**≡**c'**, então é chamado o método "*AdicioneTipo*", onde **c'** é passada como parâmetro. O método "*AdicioneTipo*", apresentado na Figura 4.12, faz a fusão do objeto **f** com **c** da seguinte forma:

1. Adiciona o nome da classe raiz que originou **c'** (**c'.ClasseDeOrigem**) na coleção **f.classes**[7]. Dessa forma, **f** passa a ser membro também de $\mathbf{C}_k$.

2. Atribui um valor para cada propriedade $\boldsymbol{p}_j$ de **f** que ainda não possui valor atribuído, mas tem um valor definido para a propriedade correspondente em **c**. Como mostrado na Figura 4.12, caso a propriedade $\boldsymbol{p}_j$ seja um atributo, então o valor da propriedade correspondente em **c'** é um literal e, portanto, pode ser atribuído diretamente como valor de $\boldsymbol{p}_j$. Por exemplo, os valores dos atributos **nome**, **salário**, **telefone**, **rg** e **nomeDepto** de EST&EMP são obtidos diretamente das propriedades correspondentes em **c'**. Caso a propriedade $\boldsymbol{p}_j$ seja um relacionamento monovalorado, então o valor da propriedade correspondente em **c'** é uma EIO, a qual é usada para fazer o "*matching*" com o objeto na visão. Por exemplo, a propriedade **curso** de EST&EMP é um relacionamento monovalorado cujo domínio é CURSO_v. Nesse caso, **c'** terá uma propriedade **curso** cujo valor é uma EIO que tem as informações necessárias para identificar o objeto correspondente da classe de visão CURSO_v, o qual é atribuído como valor da propriedade **curso** do objeto **e**. Note que a EIO de CURSO_v foi obtida no momento da atribuição dos valores às propriedades de EST&EMP realizada pelo método *GereEO_de_*EST&EMP. No caso dos relacionamentos multivalorados, a diferença é que é necessário repetir o procedimento usado para os relacionamentos monovalorados para todos os objetos da coleção. Esse é o caso da propriedade **dependentes** de EST&EMP.

Caso 2: *Não existe um objeto em* **F** *semanticamente equivalente a* **c'**

No caso de ainda não existir um objeto **f** em **F**, tal que **f**≡**c'**, então é criada uma instância de **F** a partir da especificação de objeto **c'**. Isso é feito usando o método "*CrieObjeto*" da classe de fusão **F**. O método "*CrieObjeto*", mostrado na Figura 4.13, é bastante semelhante ao método "*AdicioneTipo*", discutido anteriormente, apresentando como diferença o comando para criar um novo objeto **f** em **F**, além de não precisar testar se as propriedades de **f** possuem valor "null".

É importante notar que para cada classe raiz $\mathbf{C}_k$ de **F**, haverá um método "*GereEO_de_***F**" e um método "*GereEIO_de_***F**". No enfoque proposto, esses métodos são gerados automaticamente através de algoritmos que usam as assertivas de correspondência entre **F** e $\mathbf{C}_k$.

O método "*GereEO_de_***F**" da classe $\mathbf{C}_k$, é gerado pelo algoritmo "A_1", apresentado no apêndice A.1. Para criar o método "*GereEO_de_***F**" corretamente, o algoritmo "A_1" atribui um valor para cada propriedade $\boldsymbol{p}_i$ da especificação de objeto **c'** de **F**, os quais são

[7] Para a inserção de valores em um conjunto é usado o comando *insert_element* do padrão ODMG-97.

AdicioneTipo(Especificação_de_objeto **c'**)
/* Atribui valores para aquelas propriedades do objeto correspondentes às propriedades de **c'** e que ainda não possuem valores definidos */
{
this.**classes**.*insert_element*(**c'**.**ClasseDeOrigem**);
Para cada propriedade $\boldsymbol{p}_j \in$ **this**.**propriedades**,$1 \leq \text{j} \leq \text{n}$, faça
    Se **c'**.$\mathbf{p}_j \neq$ null e **this**.$\mathbf{p}_j$=null então
        Se $\boldsymbol{p}_j$ é um atributo então
            this.$\boldsymbol{p}_j$ = **c'**.$\mathbf{p}_j$;
        Senão   /* $\boldsymbol{p}_j$ é um relacionamento */
            Se $\mathbf{p}_j$ é monovalorado então
                Selecione o objeto **o** de $Dom(\mathbf{p}_j)$ usando **c'**.$\mathbf{p}_j$;
                this.$\boldsymbol{p}_j$ = **o**;
            Senão
                Para cada *id* em **c'**.$\mathbf{p}_j$ faça
                    Selecione o objeto **o** de $Dom(\mathbf{p}_j)$ usando *id*;
                    this.$\boldsymbol{p}_j$.*insert_element*(**o**);
}

Figura 4.12: Método *AdicioneTipo*

determinados pelas assertivas de correspondência de propriedades e de caminhos entre **F** e $\mathbf{C}_k$. Caso a propriedade $\boldsymbol{p}_i$ seja um atributo (linha 5), então o valor da propriedade correspondente em **c** (**c**.**p**) é um literal e seu valor poderá ser atribuído diretamente como valor de $\boldsymbol{p}_i$. Caso a propriedade $\boldsymbol{p}_i$ seja um relacionamento (linha 7), então o valor da propriedade correspondente em **c** (**c**.**p**) é um objeto. Nesse caso, se o relacionamento for monovalorado (linha 8), deve ser gerada uma EIO correspondente que será usada para fazer o "*matching*" com o objeto correspondente em $Dom(\boldsymbol{p}_i)$. No caso dos relacionamentos multivalorados (linha 10), a diferença é que é necessário repetir o procedimento usado para relacionamentos monovalorados para todos os objetos da coleção. É importante notar que, neste trabalho, apenas são tratados caminhos monovalorados cujas propriedades são definidas em classes de um mesmo esquema base.

Considere, por exemplo, o método "*GereEO_de_*EST&EMP", apresentado na Figura 4.11. Nesse método, a linha 1, gerada pela linha 2 do algoritmo "A_1", inicializa as variáveis correspondentes às propriedades da EO de EST&EMP atribuindo valor "null". As linhas 2-4 foram geradas pelas linhas 4-6 do algoritmo "A_1" com base nas assertivas de correspondência de propriedades Ψ_4, Ψ_6 e Ψ_9 e determinam os valores para os respectivos atributos **nome**, **salário** e **rg** de EST&EMP. As linhas 5 e 6 foram geradas pelas linhas

```
CrieObjeto(Especificação_de_objeto c');
/* Cria um objeto em F a partir da especificação de objeto c' */
{
f= new F();        /* Cria um objeto em F*/
f.classes.insert_element(c'.ClasseDeOrigem);
Para cada propriedade p_j ∈ f.propriedades,1 ≤ j ≤ n, faça
   Se c'.p_j ≠ null então
      Se p_j é um atributo então
          f.p_j = c'.p_j;
      Senão        /* p_j é um relacionamento */
         Se p_j é monovalorado então
            Selecione o objeto o de Dom(p_j) usando c'.p_j;
            f.p_j = o;
         Senão
            Para cada id em c'.p_j faça
               Selecione o objeto o de Dom(p_j) usando id;
               this.p_j.insert_element(o);
}
```

Figura 4.13: Método *CrieObjeto*

10-12 do algoritmo "A_1" com base na assertiva de correspondência de propriedades Ψ_{11} e determinam o valor para a propriedade **dependentes** de EST&EMP. Como a propriedade **dependentes** é um relacionamento multivalorado, o valor de **dependentes** é uma coleção de EIOs que serão usadas para identificar os objetos na classe de visão DEPENDENTE$_v$ (*Dom*(**dependentes**)=DEPENDENTE$_v$) que são semanticamente equivalentes aos objetos em **dependentes**$_1$. A linha 7 foi gerada pelas linhas 13-15 do algoritmo "A_1" com base na assertiva de correspondência de caminho Ψ_{12} e determina o valor para o atributo derivado **nomeDepto** de EST&EMP.

O método "*GereEIO_de_***F**" da classe $\mathbf{C}_k$ é gerado pelo algoritmo "A_2", apresentado no apêndice A.2. Como pode ser observado no apêndice A.2, o algoritmo "A_2" é bastante semelhante ao algoritmo "A_1", discutido anteriormente, apresentando como principal diferença as propriedades de **F** consideradas, que nesse caso são apenas aquelas usadas para fazer o "*matching*" dos objetos de **F** e $\mathbf{C}_k$.

• Manutenção das Assertivas de Correspondência de Propriedades

As operações relevantes para uma ACP Ψ:$\mathbf{F}.\boldsymbol{p} \equiv \mathbf{C}_k.\boldsymbol{p'}$ são as modificações dos valores da propriedade $\boldsymbol{p'}$. A assertiva de correspondência Ψ especifica que as propriedades $\boldsymbol{p}$ e

p' são sinônimas, portanto a modificação do valor da propriedade ***p'*** de um objeto **c** de $\mathbf{C}_k$ requer a modificação do valor da propriedade ***p*** de um objeto **f** de **F**, onde o objeto **f** "*match*" o objeto **c** (**f**≡**c**). Existem quatro tipos de regras para fazer a manutenção de uma ACP. A escolha da regra depende do tipo da propriedade (se é atributo ou relacionamento) e da cardinalidade da propriedade (monovalorada ou multivalorada). Essas regras são apresentadas a seguir.

Atributos monovalorados

Suponha ***p*** e ***p'*** atributos monovalorados de **F** e $\mathbf{C}_k$, respectivamente. A Figura 4.14 mostra a regra que deve ser gerada para fazer a manutenção da ACP Ψ:**F**.***p*** ≡ $\mathbf{C}_k$.***p'*** nas modificações do valor da propriedade ***p'*** de um objeto **c** de $\mathbf{C}_k$. Note que o evento dessa regra está parametrizado com o valor atribuído a propriedade ***p'*** e com o objeto **c** de $\mathbf{C}_k$ que foi modificado[8]. A ação da regra mostrada na Figura 4.14 consiste primeiramente em gerar a especificação de identificador de objeto *id* que será usada para fazer o "*matching*" do objeto **c** do banco de dados local (**c** é o objeto que foi modificado) com o objeto em **F** que é semanticamente equivalente a **c**. Para isso, é chamado o método "*GereEIO_de_***F**" da classe $\mathbf{C}_k$, como foi feito na regra apresentada na Figura 4.4. Uma vez gerada *id*, as atualizações a serem realizadas na classe **F** são então enviadas para o mediador, as quais são descritas a seguir:

Quando modificação(v, **c**) em $\mathbf{C}_k$.***p'*** /* v é o valor atribuído a **c**.***p'*** */
Então *id*= **c**.*GereEIO_de_***F**();
Requisita as seguintes atualizações para **M**:
<Selecione **f** de **F** usando *id*; /* **f**≡**c** */
f.**p**= v;>

Figura 4.14: Regra para manutenção de Ψ:**F**.***p*** ≡ $\mathbf{C}_k$.***p'*** onde ***p*** e ***p'*** são atributos

1. selecionar o objeto **f** de **F** tal que **f** é semanticamente equivalente ao objeto **c** usando *id*.
2. atribuir o valor v como valor de **f**.***p*** (**f**.**p**= v).

Considere, por exemplo, a classe de fusão completa EST&EMP apresentada na Figura 3.5. A regra gerada para fazer a manutenção da ACP Ψ_6:EST&EMP.**salário** ≡ EMPREGADO.**salário**$_1$ é mostrada na Figura 4.15.

[8] Em um sistema como o Oracle8, uma propriedade modificada ***p*** é referenciada como "*new*.**p**".

Quando modificação(v, **c**) em EMPREGADO.**salário**$_1$ /*v é o valor atribuído a **c.salário***/
Então id= **c**.*GereEIO_de_*EST&EMP();
Requisita as seguintes atualizações para **M**:
<Selecione **e** de EST&EMP usando id; /* **e**≡**c** */
e.salário= v;>

Figura 4.15: Regra para a manutenção de Ψ_6 nas atualizações em **salário**$_1$

Relacionamentos monovalorados

Suponha ***p*** e ***p'*** relacionamentos monovalorados de **F** e $\mathbf{C}_k$, respectivamente, onde $Dom(\mathbf{p})$ = **D**. A Figura 4.16 mostra a regra que deve ser gerada para fazer a manutenção da ACP Ψ:**F**.***p*** ≡ $\mathbf{C}_k$.***p'*** nas modificações do valor da propriedade ***p'*** de um objeto **c** de $\mathbf{C}_k$. A ação dessa regra consiste primeiramente em gerar a especificação de identificador de objeto id_c que será usada para fazer o "*matching*" do objeto **c** do banco de dados local (**c** é o objeto que foi modificado) com o objeto em **F** que é semanticamente equivalente a **c**. Para isso, é chamado o método "*GereEIO_de_***F**" da classe $\mathbf{C}_k$, tal como foi feito na regra mostrada na Figura 4.14. O próximo passo é gerar a especificação de identificador de objeto id_v que será usada para fazer o "*matching*" do objeto **v** da classe **V** que foi atribuído a **c**.***p'*** com o objeto em $Dom(\mathbf{p})$ que é semanticamente equivalente a **v**. Para gerar a EIO id_v, é chamado o método "*GereEIO_de_***D**" da classe base **V**. Uma vez gerada as especificações de identificador de objeto id_c e id_v, as atualizações a serem realizadas na classe **F** são então enviadas para o mediador, as quais são descritas a seguir:

Quando modificação(**v**, **c**) em $\mathbf{C}_k$.***p'*** /* **v** é o objeto atribuído a **c**.***p'*** */
Então id_c= **c**.*GereEIO_de_***F**();
id_v= **v**.*GereEIO_de_***D**(); /* $Dom(\mathbf{p})$ = **D** */
Requisita as seguintes atualizações para **M**:
<Selecione **f** de **F** usando id_c; /* **f**≡**c** */
Selecione **d** de **D** usando id_v; /* **d**≡**v** */
f.***p***= **d**;>

Figura 4.16: Regra para manutenção de Ψ:**F**.***p*** ≡ $\mathbf{C}_k$.***p'*** onde ***p*** e ***p'*** são relacionamentos

1. selecionar o objeto **f** de **F** tal que **f** é semanticamente equivalente ao objeto **c** usando id_c.

2. selecionar o objeto **d** de **D** tal que **d** é semanticamente equivalente ao objeto **v** usando id_v.

3. atribuir o objeto **d** como valor de **f**.***p*** (**f**.***p***= **d**).

Na Figura 4.17 é mostrada a regra gerada para fazer a manutenção da ACP Ψ_7: EST&EMP.**curso** ≡ ESTUDANTE.**curso**$_2$.

Quando modificação(**v**, **c**) em ESTUDANTE.**curso**$_2$ /* **v** é o objeto atribuído a **c**.**curso**$_2$ */
Então id_c= **c**.*GereEIO_de_*EST&EMP();
id_v= **v**.*GereEIO_de_*CURSO$_v$(); /* *Dom*(**curso**) = CURSO$_v$ */
Requisita as seguintes atualizações para **M**:
<Selecione **e** de EST&EMP usando id_c; /* **e**≡**c** */
Selecione **v'** de CURSO$_v$ usando id_v; /* **v'**≡**v** */
e.**curso**= **v'**;>

Figura 4.17: Regra para a manutenção de Ψ_7 nas atualizações em **curso**$_2$

Atributos multivalorados

Suponha ***p*** e ***p'*** atributos multivalorados de **F** e $\mathbf{C}_k$, respectivamente, e suponha **c** o objeto de $\mathbf{C}_k$ que foi modificado. No caso dos atributos multivalorados, o valor de uma propriedade é uma coleção, podendo este ser modificado através das operações de inserção e remoção de elementos da coleção. Assim sendo, para uma dada assertiva de correspondência Ψ:**F**.***p*** ≡ $\mathbf{C}_k$.***p'*** deve ser gerada uma regra para inserções e outra para remoções de valores em **c**.***p'*** de $\mathbf{C}_k$, as quais são descritas a seguir.

Regra para inserções na coleção

A Figura 4.18 mostra a regra que deve ser gerada para fazer a manutenção da ACP Ψ:**F**.***p*** ≡ $\mathbf{C}_k$.***p'*** nas inserções em **c**.***p'***. A ação dessa regra é semelhante a da regra mostrada na Figura 4.14 e consiste primeiramente em gerar a especificação de identificador de objeto *id* que será usada para fazer o "*matching*" do objeto **c** com o objeto em **F** que é semanticamente equivalente a **c**. Para isso, é chamado o método "*GereEIO_de_***F**" da classe $\mathbf{C}_k$. Uma vez gerada *id*, as atualizações a serem realizadas na classe **F** são então enviadas para o mediador, as quais são descritas a seguir:

1. selecionar o objeto **f** de **F** tal que **f** é semanticamente equivalente ao objeto **c** usando *id*.

2. inserir o valor v na coleção **f**.***p*** (**f**.***p***.*insert_element*(v)).

Quando inserção(v, **c**) em $\mathbf{C}_k.\boldsymbol{p'}$ /* v é o valor inserido em $\mathbf{c}.\boldsymbol{p'}$ */
Então *id*= **c**.*GereEIO_de_***F**();
Requisita as seguintes atualizações para **M**:
<Selecione **f** de **F** usando *id*; /* **f**≡**c** */
f.**p**.*insert_element*(v);>

Figura 4.18: Regra para manutenção de Ψ nas inserções em $\mathbf{c}.\boldsymbol{p'}$, onde $\boldsymbol{p'}$ é um atributo

Na Figura 4.19, é mostrada a regra gerada para fazer a manutenção da ACP Ψ_8:EST&EMP.**telefone** ≡ ESTUDANTE.$\mathbf{telefone}_2$ nas inserções em $\mathbf{c}.\mathbf{telefone}_2$, onde **c** é um objeto de ESTUDANTE.

Quando inserção(v, **c**) em ESTUDANTE.$\mathbf{telefone}_2$ /* v é o valor inserido em $\mathbf{c}.\mathbf{telefone}_2$ */
Então *id*= **c**.*GereEIO_de_*EST&EMP();
Requisita as seguintes atualizações para **M**:
<Selecione **e** de EST&EMP usando *id*; /* **e**≡**c** */
e.**telefone**.*insert_element*(v);>

Figura 4.19: Regra para a manutenção de Ψ_8 nas inserções em $\mathbf{c}.\mathbf{telefone}_2$

Regras para remoções da coleção

A Figura 4.20 mostra a regra que deve ser gerada para fazer a manutenção da ACP Ψ:$\mathbf{F}.\boldsymbol{p} \equiv \mathbf{C}_k.\boldsymbol{p'}$ nas remoções em $\mathbf{c}.\boldsymbol{p'}$. A ação dessa regra é semelhante a da regra mostrada na Figura 4.18; a única diferença está na operação para atualizar a propriedade $\boldsymbol{p}$, que é "*remove_element*" no caso da remoção de valores da coleção.

Quando remoção(v, **c**) em $\mathbf{C}_k.\boldsymbol{p'}$ /* v é o valor removido de $\mathbf{c}.\boldsymbol{p'}$ */
Então *id*= **c**.*GereEIO_de_***F**();
Requisita as seguintes atualizações para **M**:
<Selecione **f** de **F** usando *id*; /* **f**≡**c** */
f.**p**.*remove_element*(v);>

Figura 4.20: Regra para manutenção de Ψ nas remoções em $\mathbf{c}.\boldsymbol{p'}$, onde $\boldsymbol{p'}$ é um atributo

Na Figura 4.21, é mostrada a regra gerada para fazer a manutenção da ACP Ψ_8:EST&EMP.**telefone** ≡ ESTUDANTE.$\mathbf{telefone}_2$ para as remoções em $\mathbf{c}.\mathbf{telefone}_2$ de ESTUDANTE.

Quando remoção(v, **c**) em ESTUDANTE.**telefone**$_2$ /*v é o valor removido de **c.telefone**$_2$*/
Então *id*= **c**.*GereEIO_de_*EST&EMP();
Requisita as seguintes atualizações para **M**:
<Selecione **e** de EST&EMP usando *id*; /* **e**≡**c** */
e.telefone.*remove_element*(v);>

Figura 4.21: Regra para manutenção de Ψ_8 nas remoções em **c.telefone**$_2$

Relacionamentos multivalorados

Suponha ***p*** e ***p'*** relacionamentos multivalorados de **F** e $\mathbf{C}_k$, respectivamente, onde $Dom(\mathbf{p})$ = **D** e suponha **c** o objeto de $\mathbf{C}_k$ que foi modificado. No caso dos relacionamentos multivalorados, o valor de uma propriedade é uma coleção de objetos, podendo este ser modificado através das operações de inserção e remoção de objetos da coleção. Assim sendo, para uma dada assertiva de correspondência Ψ:**F**.***p*** $\equiv$ $\mathbf{C}_k$.***p'*** deve ser gerada uma regra para inserções e outra para remoções de objetos em **c**.***p'*** de $\mathbf{C}_k$, as quais são descritas a seguir.

Regras para inserções na coleção

A Figura 4.22 mostra a regra que deve ser gerada para fazer a manutenção da ACP Ψ:**F**.***p*** $\equiv$ $\mathbf{C}_k$.***p'*** nas inserções em **c**.***p'***. A ação dessa regra é semelhante a regra mostrada na Figura 4.16 e consiste primeiramente em gerar a especificação de identificador de objeto id_c que será usada para fazer o "*matching*" do objeto **c** com o objeto em **F** que é semanticamente equivalente a **c**. Para isso, é chamado o método "*GereEIO_de_***F**" da classe $\mathbf{C}_k$. O próximo passo é gerar a especificação de identificador de objeto id_v que será usada para fazer o "*matching*" do objeto **v** da classe **V** que foi atribuído a **c**.***p'*** com o objeto em $Dom(\mathbf{p})$ que é semanticamente equivalente a **v**. Para gerar a EIO id_v, é chamado o método "*GereEIO_de_***D**" da classe base **V**. Uma vez gerada as especificações de identificador de objeto id_c e id_v, as atualizações a serem realizadas na classe **F** são então enviadas para o mediador, as quais são descritas a seguir:

1. selecionar o objeto **f** de **F** tal que **f** é semanticamente equivalente ao objeto **c** usando id_c.

2. selecionar o objeto **d** de **D** tal que **d** é semanticamente equivalente ao objeto **v** usando id_v.

3. inserir o objeto **d** na coleção **f**.***p*** (**f**.***p***.*insert_element*(**d**)).

Quando inserção(**v**, **c**) em $\mathbf{C}_k$.***p'*** /* **v** é o objeto inserido em **c**.***p'*** */
Então id_c= **c**.*GereEIO_de_***F**();
id_v= **v**.*GereEIO_de_***D**(); /* *Dom*(**p**)= **D** */
Requisita as seguintes atualizações para **M**:
<Selecione **f** de **F** usando id_c; /* **f**≡**c** */
Selecione **d** de *Dom*(**p**) usando id_v; /* **d**≡**v** */
f.**p**.*insert_element*(**d**)>;

Figura 4.22: Regra para manutenção de Ψ nas inserções em **c**.***p'***, onde ***p'*** é um relacionamento

Na Figura 4.23, é mostrada a regra gerada para fazer a manutenção da ACP Ψ_{11}:EST&EMP.**dependentes** ≡ EMPREGADO.$\mathbf{dependentes}_1$ para as inserções em **c**.$\mathbf{dependentes}_1$, onde **c** é um objeto de EMPREGADO.

Quando inserção(**v**, **c**) em EMPREGADO.$\mathbf{dependentes}_1$ /* **v** é o objeto inserido em */
Então id_c= **c**.*GereEIO_de_*EST&EMP(); /* **c**.$\mathbf{dependentes}_1$ */
id_v= **v**.*GereEIO_de_*DEPENDENTE_v(); /* *Dom*(**dependentes**)= DEPENDENTE_v */
Requisita as seguintes atualizações para **M**:
<Selecione **e** de EST&EMP usando id_c; /* **e**≡**c** */
Selecione **v'** de DEPENDENTE_v usando id_v; /* **v'**≡**v** */
e.**dependentes**.*insert_element*(**v'**);>

Figura 4.23: Regra para manutenção de Ψ_{11} nas inserções em **c**.$\mathbf{dependentes}_1$

Regras para remoções da coleção

A Figura 4.24 mostra a regra que deve ser gerada para fazer a manutenção da ACP Ψ:**F**.***p*** ≡ $\mathbf{C}_k$.***p'*** nas remoções em **c**.***p'***. A ação dessa regra é semelhante a da regra mostrada na Figura 4.22; a única diferença está na operação para atualizar a propriedade **p**, que é "*remove_element*" no caso da remoção de objetos da coleção.

Na Figura 4.25, é mostrada a regra gerada para fazer a manutenção da ACP Ψ_{11}:EST&EMP.**dependentes** ≡ EMPREGADO.$\mathbf{dependentes}_1$ para as remoções em **c**.$\mathbf{dependentes}_1$ de EMPREGADO.

Quando remoção(**v**, **c**) em $\mathbf{C}_k.\boldsymbol{p'}$ /* **v** é o valor removido de $\mathbf{c}.\boldsymbol{p'}$ */
Então id_c= **c**.*GereEIO_de_***F**();
id_v= **v**.*GereEIO_de_***D**(); /* *Dom*(**p**)= **D** */
Requisita as seguintes atualizações para **M**:
<Selecione **f** de **F** usando id_c; /* **f**≡**c** */
Selecione **d** de *Dom*(**p**) usando id_v; /* **d**≡**v** */
f.**p**.*remove_element*(**d**);>

Figura 4.24: Regra para manutenção de Ψ nas remoções em $\mathbf{c}.\boldsymbol{p'}$, onde $\boldsymbol{p'}$ é um relacionamento

• Manutenção das Assertivas de Correspondência de Caminhos Monovalorados

Nesta seção, considere a ACC Ψ:$\mathbf{F}.\boldsymbol{p} \equiv \mathbf{R}_1.\boldsymbol{p}_1 \bullet \ldots \bullet \boldsymbol{p}_q$, onde $\boldsymbol{p}_i$ é uma propriedade da classe base $\mathbf{R}_i$, para $1 \leq i \leq q$, e $\mathbf{R}_1$ é uma classe raiz de **F**. A assertiva de correspondência Ψ especifica que a propriedade $\boldsymbol{p}$ e o caminho $\varrho = \boldsymbol{p}_1 \bullet \ldots \bullet \boldsymbol{p}_q$ são semanticamente equivalentes. As operações relevantes para a ACC Ψ são as modificações no valor de uma propriedade $\boldsymbol{p}_i$ de um objeto **r** da classe $\mathbf{R}_i$. Assim, uma regra deve ser gerada para modificações de cada propriedade $\boldsymbol{p}_i$ de ϱ. As regras geradas diferem de acordo com o tipo do caminho, o qual pode ser de valor ou de referência, e são apresentadas a seguir.

Caminhos de valor

Suponha $\boldsymbol{p}$ um atributo monovalorado e $\varrho = \boldsymbol{p}_1 \bullet \ldots \bullet \boldsymbol{p}_q$ um caminho de valor monovalorado. Como definido na seção 3.2, o valor de um caminho de valor é um literal, ou seja, o $Dom(\boldsymbol{p}_q)$ é uma classe literal. A Figura 4.26 mostra a regra que deve ser gerada para fazer a manutenção da ACC Ψ nas modificações do valor da propriedade $\boldsymbol{p}_i$ de um objeto **r** da classe $\mathbf{R}_i$. A ação dessa regra consiste primeiramente em realizar as seguintes operações

Quando remoção(**v**,**c**) em EMPREGADO.$\mathbf{dependentes}_1$ /* **v** é o objeto removido de */
Então id_c= **c**.*GereEIO_de_*EST&EMP(); /* $\mathbf{c}.\mathbf{dependentes}_1$ */
id_v= **v**.*GereEIO_de_*DEPENDENTE_v(); /* *Dom*(**dependentes**)= DEPENDENTE_v */
Requisita as seguintes atualizações para **M**:
<Selecione **e** de EST&EMP usando id_c; /* **e**≡**c** */
Selecione **v'** de DEPENDENTE_v usando id_v; /* **v'**≡**v** */
e.**dependentes**.*remove_element*(**v'**);>

Figura 4.25: Regra para manutenção de Ψ_{11} nas remoções em $\mathbf{c}.\mathbf{dependentes}_1$

no banco de dados local:

Quando modificação(v, $\mathbf{r}$) em $\mathbf{R}_i.\boldsymbol{p}_i$ /* v é o valor atribuído a $\mathbf{r}.\boldsymbol{p}_i$ */

Então Selecione os objetos $\mathbf{c}_1, ..., \mathbf{c}_n$ de $\mathbf{R}_1$ tais que $\mathbf{c}_j.\boldsymbol{p}_1 \bullet ... \bullet \boldsymbol{p}_q == v$, para $1 \leq \mathrm{j} \leq \mathrm{n}$;
Para cada $\mathbf{c}_j$, $1 \leq \mathrm{j} \leq \mathrm{n}$, faça
$id_j = \mathbf{c}_j.GereEIO_de_\mathbf{F}()$;
$v' = \mathbf{c}_j.\boldsymbol{p}_1 \bullet ... \bullet \boldsymbol{p}_q$
Requisita as seguintes atualizações para $\mathbf{M}$:
<Para cada id_j, $1 \leq \mathrm{j} \leq \mathrm{n}$, faça
Selecione $\mathbf{f}$ de $\mathbf{F}$ usando id_j; /* $\mathbf{f} \equiv \mathbf{c}_j$ */
$\mathbf{f}.\mathbf{p} = v'$;>

Figura 4.26: Regra para manutenção de Ψ nas modificações em $\mathbf{r}.\boldsymbol{p}_i$, onde $\boldsymbol{p}_i$ é uma das propriedades de um caminho de valor

1. selecionar o conjunto de objetos da classe $\mathbf{R}_1$ relacionados ao objeto $\mathbf{r}$. É importante notar que mais de um objeto de $\mathbf{R}_1$ podem estar relacionados com o objeto $\mathbf{r}$, isto acontece quando o caminho inverso de ϱ é multivalorado.

2. Para cada objeto $\mathbf{c}_j$ relacionado ao objeto $\mathbf{r}$, $1 \leq \mathrm{j} \leq \mathrm{n}$, é gerada uma especificação de identificador de objeto id_j, a qual será usada para fazer o "*matching*" do objeto $\mathbf{c}_j$ com o objeto de $\mathbf{F}$ que é semanticamente equivalente a $\mathbf{c}_j$.

3. obter o novo valor v' do caminho ϱ ($v' = \mathbf{c}_j.\boldsymbol{p}_1 \bullet ... \bullet \boldsymbol{p}_q$).

Uma vez executadas essas operações, são então enviadas para o mediador as seguintes atualizações:

- atribuir v' como valor de $\boldsymbol{p}$ para todos os objetos $\mathbf{f}$ de $\mathbf{F}$ que são semanticamente equivalentes as EIOs geradas.

Considere, por exemplo, a classe de fusão completa EST&EMP apresentada na Figura 3.5. A regra gerada para fazer a manutenção da ACC Ψ_{12}:EST&EMP.**nomeDepto**≡EMPREGADO.**depto**$_1$ • **nome**$_1$ nas modificações em **d**.**nome**$_1$, onde **d** é um objeto de DEPARTAMENTO, é mostrada na Figura 4.27.

Quando modificação(v, **d**) em DEPARTAMENTO.**nome**$_1$ /*v é o valor atrib. a **d**.**nome**$_1$ */
Então Selecione os objetos $\mathbf{e}_1$, ..., $\mathbf{e}_n$ de EMPREGADO tais que
$\mathbf{e}_j$.**depto**$_1$ • **nome**$_1$==v, para $1 \leq j \leq n$;
Para cada $\mathbf{e}_j$, $1 \leq j \leq n$, faça
id_j= $\mathbf{e}_j$.*GereEIO_de*_EST&EMP();
v'= $\mathbf{e}_j$.**depto**$_1$ • **nome**$_1$;
Requisita as seguintes atualizações para **M**:
<Para cada id_j, $1 \leq j \leq n$, faça
Selecione **e** de EST&EMP usando id_j; /* $\mathbf{e} \equiv \mathbf{e}_j$ */
e.**nomeDepto**= v';>

Figura 4.27: Regra para manutenção de Ψ_{12} nas modificações em **d**.**nome**$_1$

Caminhos de referência

Suponha $\boldsymbol{p}$ um relacionamento monovalorado e ϱ=$\boldsymbol{p}_1$ • ... • $\boldsymbol{p}_q$ um caminho de referência monovalorado. Como definido na seção 3.2, o valor de um caminho de referência é um objeto, ou seja, o $Dom(\boldsymbol{p}_q)$ é uma classe de objeto. A Figura 4.28 mostra a regra que deve ser gerada para fazer a manutenção da ACC Ψ nas modificações do valor da propriedade $\boldsymbol{p}_i$ de um objeto **r** da classe $\mathbf{R}_i$. A ação dessa regra é semelhante a da regra mostrada na Figura 4.26 e consiste primeiramente em realizar as seguintes operações no banco de dados local:

1. selecionar o conjunto de objetos da classe $\mathbf{R}_1$ relacionados ao objeto **r**.

2. Para cada objeto $\mathbf{c}_j$ relacionado ao objeto **r**, $1 \leq j \leq n$, é gerada uma especificação de identificador de objeto id_j, a qual será usada para fazer o "*matching*" do objeto $\mathbf{c}_j$ com o objeto de **F** que é semanticamente equivalente a $\mathbf{c}_j$.

3. obter o objeto **v** correspondente ao novo valor do caminho ϱ (**v**= $\mathbf{c}_j.\boldsymbol{p}_1$ •... • $\boldsymbol{p}_q$).

4. gerar a especificação de identificador de objeto id_v do objeto **v**, a qual será usada para fazer o "*matching*" do objeto **v** com o objeto **d** de $Dom(\mathbf{p})$.

Uma vez executadas essas operações, são então enviadas para o mediador as seguintes atualizações:

1. selecionar o objeto **d** de $Dom(\mathbf{p})$ tal que **d** é semanticamente equivalente ao objeto **v** usando id_v.

2. atribuir o objeto **d** como valor de $\boldsymbol{p}$ para todos os objetos **f** de **F** que são semanticamente equivalentes as EIOs de **F** geradas.

Quando modificação($\mathbf{v}$, $\mathbf{r}$) em $\mathbf{R}_i.\boldsymbol{p}_i$ /* v é o valor atribuído a $\mathbf{r}.\boldsymbol{p}_i$ */
Então Selecione os objetos $\mathbf{c}_1, ..., \mathbf{c}_n$ de $\mathbf{R}_1$ tais que $\mathbf{c}_j.\boldsymbol{p}_1 \bullet ... \bullet \boldsymbol{p}_q$== $\mathbf{v}$, para $1 \leq$ j $\leq$ n;
Para cada $\mathbf{c}_j$, $1 \leq$ j $\leq$ n, faça
id_j= $\mathbf{c}_j$.*GereEIO_de_*$\mathbf{F}$();
$\mathbf{v}$= $\mathbf{c}_j.\boldsymbol{p}_1 \bullet ... \bullet \boldsymbol{p}_q$;
id_v= $\mathbf{v}$.*GereEIO_de_*$\mathbf{D}$(); /* *Dom*($\mathbf{p}$)= $\mathbf{D}$ */
Requisita as seguintes atualizações para $\mathbf{M}$:
Selecione $\mathbf{d}$ de *Dom*($\mathbf{p}$) usando id_v;
<Para cada id_j, $1 \leq$ j $\leq$ n, faça
Selecione $\mathbf{f}$ de $\mathbf{F}$ usando id_j; /* $\mathbf{f} \equiv \mathbf{c}_j$ */
$\mathbf{f}.\mathbf{p}$= $\mathbf{d}$;>

Figura 4.28: Regra para manutenção de Ψ nas modificações em $\mathbf{r}.\boldsymbol{p}_i$ onde $\boldsymbol{p}_i$ é uma das propriedades de um caminho de referência

No próximo capítulo, será mostrado como as classes de fusão completas podem ser utilizadas para permitir a auto-manutenção de outras classes de fusão que preservam os objetos.

Capítulo 5

Usando Classes de Fusão Completas para Suportar a Auto-manutenção de Classes de Visão que Preservam os Objetos

5.1 Introdução

Neste capítulo é proposta uma estratégia para o problema de auto-manutenção de classes de visão preservadoras de objetos (*object-preserving views*[55]). Como mencionado anteriormente, em ambientes de múltiplos bancos de dados, uma classe de visão preserva objetos quando existe um mapeamento 1-1 entre os objetos da classe de visão e os objetos das classes base, onde esses objetos representam uma mesma entidade do mundo real. É importante notar que as classes de visão de união, interseção, diferença, seleção, projeção (ou *hide*[48]), alguns tipos de classes de junção e de fusão, preservam os objetos. Outro tipo comum de classes que preservam os objetos são aquelas com propriedades e métodos adicionais em relação a classe base correspondente. No modelo adotado pelo projeto *MultiView*[48], por exemplo, essas classes são criadas pela operação *refine*.

Este capítulo é organizado do seguinte modo:

- Na seção 5.2, é apresentado o enfoque proposto para permitir a auto-manutenção de classes de visão preservadoras de objetos usando classes de fusão completas como auxiliares.

- Na seção 5.3, é definido como gerar a classe auxiliar para uma dada classe de visão a fim de permitir a auto-manutenção das classes de visão.

- Na seção 5.4, é apresentado como definir as classes de visão a partir das classes de visão auxiliares.
- Na seção 5.5, são mostradas as regras que devem ser geradas para fazer a manutenção das classes de visão quando estas forem materializadas.

5.2 A Arquitetura Proposta para a Auto-Manutenção de Classes de Visão

Para fazer a manutenção incremental de classes de visão que preservam os objetos e sem necessitar acessar as fontes de informação, é proposta a arquitetura mostrada na Figura 5.1.

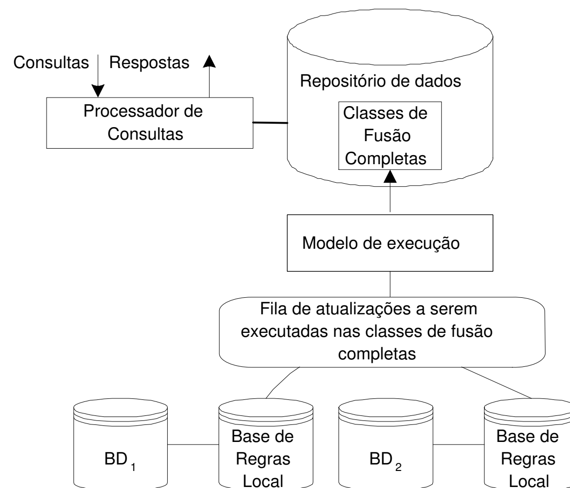

Figura 5.1: Arquitetura proposta para a auto-manutenção de classes de visão

A arquitetura proposta nesta seção é uma extensão da arquitetura apresentada na seção 4.3, diferenciando desta primeira nos seguintes aspectos:

- Usa classes de fusão completa como classes auxiliares para permitir a auto-manutenção das classes de visão preservadoras de objetos. Essas classes auxiliares armazenam os objetos das classes de visão originais já em uma forma "*sintetizada*". Isso significa que as informações de todos os objetos dos bancos de dados locais que representam uma mesma entidade do mundo real e que são membros (de fato ou em potencial) das classes de visão originais são sintetizados em um único objeto da classe auxiliar. Essas classes auxiliares, como foi mostrado na seção 4.2, são auto-manuteníveis e ficam armazenadas no repositório de dados do sistema global.

- Como será mostrado na seção 5.4, as extensões das classes de visão originais são um subconjunto das extensões das classes auxiliares correspondentes. Uma vez que as extensões das classes auxiliares são materializadas, uma classe de visão original pode ser definida virtualmente como uma simples seleção. A classe de visão original também pode ser implementada como uma visão materializada. Nesse caso, devem ser geradas regras que propagam atualizações da classe auxiliar em atualizações da classe de visão de forma a tornar a classe de visão consistente com a classe auxiliar e por conseguinte com relação aos bancos de dados locais.

- Caso as classes de visão originais sejam materializadas, então existirá no sistema global uma base de regras global, a qual armazena as regras (denominadas *regras globais*) para a manutenção das classes de visão originais.

A manutenção de uma visão de integração usando a arquitetura proposta é realizada do seguinte modo:

- Quando uma atualização relevante τ ocorre em um banco de dados local **D**, uma regra local é disparada, a qual propaga para FA as atualizações que devem ser realizadas nas classes auxiliares afetadas pela atualização τ, a fim de torná-las consistentes com o novo estado do banco de dados **D**.

- Periodicamente, o módulo de execução processa as atualizações da fila atualizando desse modo as classes auxiliares.

- Caso as classes de visão originais sejam materializadas, regras globais são disparadas quando as classes auxiliares forem atualizadas, a fim de tornar as classes de visão consistentes com as classes auxiliares e consequentemente com o novo estado do banco de dados **D**.

Nas próximas seções serão discutidos como gerar as classes auxiliares, as classes de visão originais a partir das classes auxiliares e as regras globais.

5.3 Definindo Classes Auxiliares

Nesta seção considere **V** uma classe de visão tal que **V** preserva os objetos e herda propriedades das classes raízes $\mathbf{C}_1$, ..., $\mathbf{C}_m$ e de mais nenhuma outra. A classe auxiliar **A**, a qual é uma classe de fusão completa, para a classe de visão **V** é definida como se segue:

- A extensão de **A** é definida pela assertiva de correspondência $\mathbf{A} \equiv \mathbf{C}_1 \cup ... \cup \mathbf{C}_m$. Assim sendo, as classes $\mathbf{C}_1$, ..., $\mathbf{C}_m$ são classes raízes de **A**.

- As propriedades de **A** são definidas como se segue:

  1. Para cada propriedade ***p'*** de **V**, é definida uma propriedade ***p*** em **A** tal que $\mathbf{A}.\boldsymbol{p} \equiv \mathbf{V}.\boldsymbol{p}'$ de forma que para cada assertiva de correspondência de propriedade ou de caminho envolvendo ***p'*** será definida uma assertiva de correspondência correspondente substituindo-se ***p'*** por **p**.
  2. Para cada propriedade ***p'*** de $\mathbf{C}_k$, $1 \leq \mathrm{k} \leq \mathrm{m}$, que é necessária para determinar a equivalência semântica entre objetos ou usada em um predicado para selecionar os membros da classe de visão **V**, é definida uma propriedade ***p*** em **A** tal que $\mathbf{A}.\boldsymbol{p} \equiv \mathbf{C}_k.\boldsymbol{p}'$ ou, no caso do valor requerido ser de classes relacionadas a $\mathbf{C}_k$, $\mathbf{A}.\boldsymbol{p} \equiv \mathbf{C}_k.\boldsymbol{p}' \bullet \varrho$, onde ϱ é um caminho.
  3. A propriedade **classes** tal como foi definida na seção 4.2.

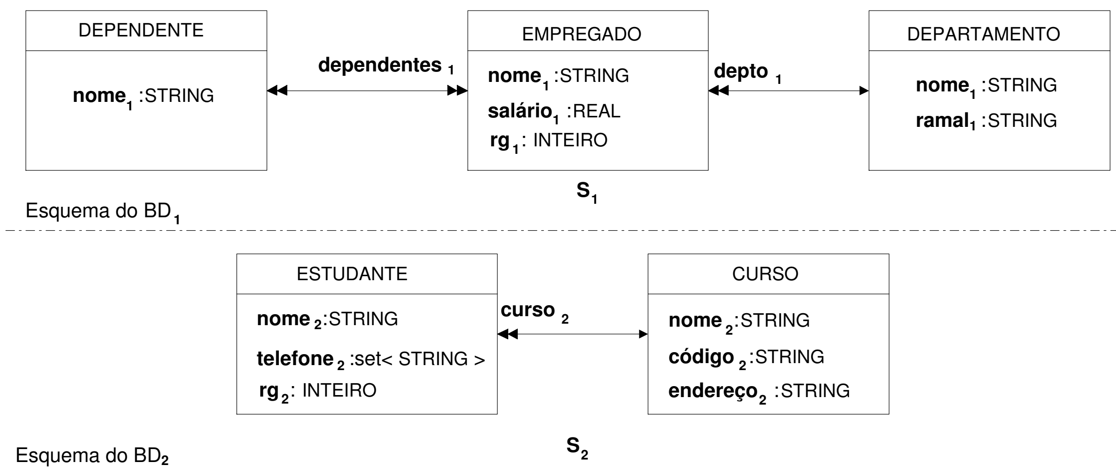

Figura 5.2: Esquemas locais S_1 e S_2

Considere, por exemplo, os esquemas dos bancos de dados locais $\mathbf{S}_1$ e $\mathbf{S}_2$, apresentados na Figura 5.2, e os esquemas das visões $\mathbf{S}_{v1}$ e $\mathbf{S}_{v2}$, apresentados na Figura 5.3. Como pode ser observado das assertivas de correspondência mostradas na Figura 5.3, a classe de visão ALUNO$_v$ tem como classes raízes ESTUDANTE e EMPREGADO. A Figura 5.4 mostra a classe auxiliar EST&EMP criada para suportar a auto-manutenção da classe de visão ALUNO$_v$. Na Figura 5.4, também são mostradas as assertivas de correspondência entre EST&EMP e suas classes raízes (ESTUDANTE e EMPREGADO).

Um fato a ser observado é que classes de visão com classes raízes em comum podem compartilhar a mesma classe auxiliar. Esse é o caso, por exemplo, das classes de

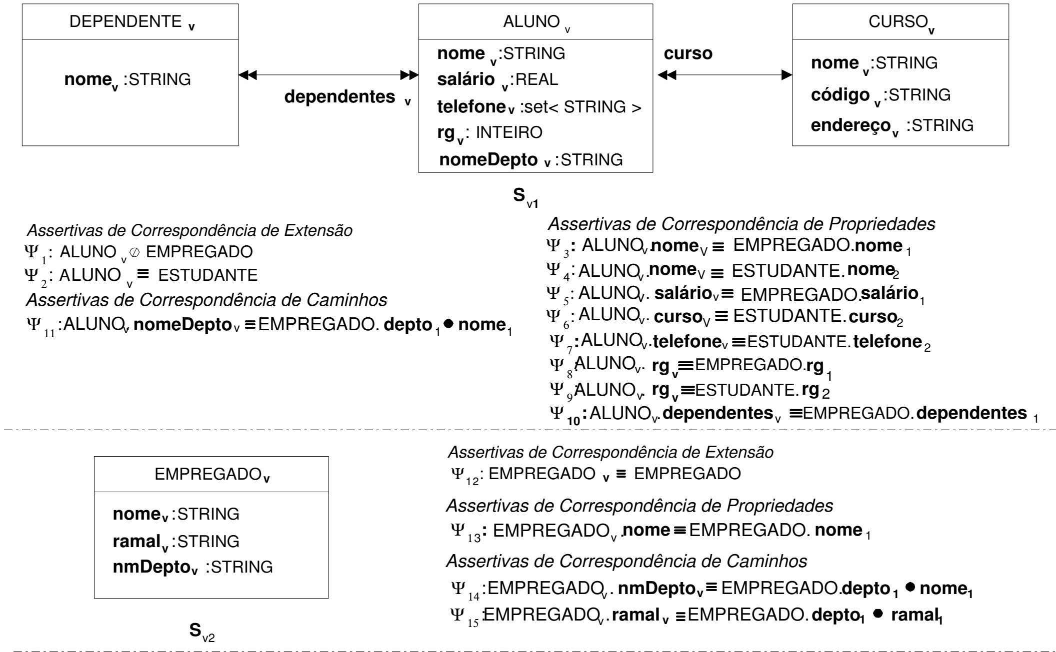

Figura 5.3: Esquema de visões S_{v1} e S_{v2} com ACs

visão EMPREGADO$_v$ e ALUNO$_v$ dos esquemas $\mathbf{S}_{v1}$ e $\mathbf{S}_{v2}$ da Figura 5.3. Como a classe raiz EMPREGADO de EMPREGADO$_v$ é também uma classe raiz de ALUNO$_v$, a classe auxiliar EST&EMP (Figura 5.4) criada para ALUNO$_v$ também pode ser usada por EMPREGADO$_v$. É importante notar, entretanto, que as propriedades de EMPREGADO$_v$ que não tem uma correspondente em ALUNO$_v$ (e portanto em EST&EMP) devem ser acrescentadas as propriedades de EST&EMP[1]. Também devem ser acrescentadas às propriedades de EST&EMP as propriedades de EMPREGADO$_v$ necessárias para determinar a equivalência semântica entre objetos ou usadas em um predicado para selecionar os membros da classe de visão EMPREGADO$_v$[2]. Para aquelas propriedades de EMPREGADO$_v$ que já tem propriedades correspondentes em EST&EMP, devem ser definidas apenas as assertivas de correspondência de propriedades correspondentes. A Figura 5.5 mostra a classe EST&EMP adaptada para ser compartilhada por ALUNO$_v$ e EMPREGADO$_v$.

Outra nota importante é que podem existir classes de visão definidas a partir de outras classes de visão. Nesse caso, apenas uma classe auxiliar deve ser criada, a qual será formada por todas as classes raízes das classes de visão envolvidas. Por exemplo, sejam as classes de visão $\mathbf{V}_1$ e $\mathbf{V}_2$ especificadas pelas assertivas de correspondência: $\mathbf{V}_1 \equiv \mathbf{C}_1$ - $\mathbf{C}_2$

[1]Como foi feito no item 2 da definição das propriedades de **A**.
[2]Como foi feito no item 3 da definição das propriedades de **A**.

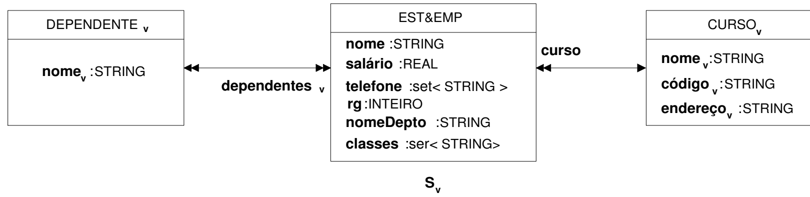

Figura 5.4: Classe auxiliar EST&EMP para a classe de visão ALUNO$_v$

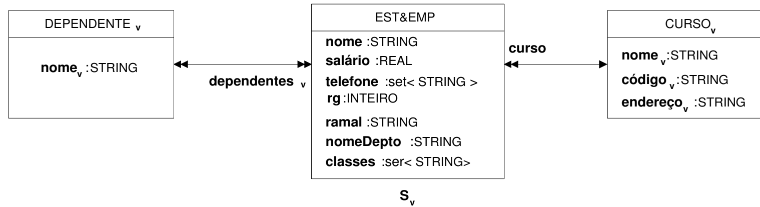

Figura 5.5: Classe auxiliar EST&EMP para as classes de visão ALUNO$_v$ e EMPREGADO$_v$

e $\mathbf{V}_2 \equiv \mathbf{V}_1 \cup \mathbf{C}_3$, em que $\mathbf{C}_1$, $\mathbf{C}_2$ e $\mathbf{C}_3$ são classes raízes. A classe auxiliar criada para as classes de visão $\mathbf{V}_1$ e $\mathbf{V}_2$ é especificada pela assertiva de correspondência $\mathbf{A} \equiv \mathbf{C}_1 \cup \mathbf{C}_2 \cup \mathbf{C}_3$. Essa assertiva de correspondência é obtida de $\mathbf{V}_2 \equiv \mathbf{V}_1 \cup \mathbf{C}_3$, substituindo-se $\mathbf{V}_1$ por sua assertiva de correspondência correspondente. Vale ressaltar que todas as propriedades das classes de visão $\mathbf{V}_1$ e $\mathbf{V}_2$ devem ter uma correspondente em $\mathbf{A}$, como mencionado anteriormente.

Uma vez definida a classe auxiliar, é preciso: 1) gerar as regras para mantê-la, conforme foi mostrado na seção 4.4; 2) redefinir as classes de visão originais usando apenas as classes auxiliares. A redefinição das classes de visão é necessária pois assim a extensão da classe de visão original pode ser obtida diretamente a partir da classe auxiliar.

5.4 Definindo Classes de Visão a partir das Classes Auxiliares

Nesta seção considere **V** uma classe de visão tal que **V** preserva os objetos e herda propriedade das classes raízes $\mathbf{C}_1$, ..., $\mathbf{C}_m$ e de mais nenhuma outra. Considere também **A** a classe auxiliar gerada, tal como definida na seção 5.3, para permitir a auto-manutenção de **V**.

Como mencionado anteriormente, a classe de visão **V** pode ser implementada como uma classe virtual ou materializada. No caso de **V** ser virtual, a definição dessa classe é determinada a partir de **A**, como será mostrado mais adiante. No caso da classe de visão ser materializada, é criada uma extensão para essa classe explicitamente. Nesse caso, devem ser geradas regras que propagam atualizações da classe auxiliar em atualizações da classe de visão. Isso é feito de forma a tornar a classe de visão consistente com a classe auxiliar e por conseguinte com relação às fontes de informação locais.

Na tabela 5.1, é definido como mapear as assertivas de correspondência de extensão entre **V** e as classes locais em assertivas de correspondência entre **V** e a classe auxiliar **A**. Com isso é possível definir as consultas de **V** a partir das assertivas de correspondência, no caso de **V** ser virtual, ou gerar as regras que determinam como manter **V** a partir de **A**, no caso de **V** ser materializada, como será mostrado na seção 5.5.

Assertivas de Correspondência entre V e as classes raízes	**Assertivas de Correspondência entre V e a classe A**
$\mathbf{V} \equiv \mathbf{C}_1$	$\mathbf{V} \equiv \mathbf{A}[\text{“}\mathbf{C}_1\text{”} \in classes]$
$\mathbf{V} \equiv \mathbf{C}_1[\mathsf{p}]$	$\mathbf{V} \equiv \mathbf{A}[\mathsf{p} \wedge \text{“}\mathbf{C}_1\text{”} \in classes]$
$\mathbf{V} \equiv \mathbf{C}_1\text{-}\mathbf{C}_2$	$\mathbf{V} \equiv \mathbf{A}[\text{“}\mathbf{C}_1\text{”} \in classes \wedge \text{“}\mathbf{C}_2\text{”} \notin classes]$
$\mathbf{V} \equiv \cup_{i=1}^{n} \mathbf{C}_i$	$\mathbf{V} \equiv \mathbf{A}[\text{“}\mathbf{C}_1\text{”} \in classes \vee \ldots \vee \text{“}\mathbf{C}_v\text{”} \in classes]$
$\mathbf{V} \equiv \cap_{i=1}^{n} \mathbf{C}_i$	$\mathbf{V} \equiv \mathbf{A}[\text{“}\mathbf{C}_1\text{”} \in classes \wedge \ldots \wedge \text{“}\mathbf{C}_v\text{”} \in classes]$

Tabela 5.1: Mapeamento entre Assertivas de Correspondência de Extensão

Considere, por exemplo, a classe de visão ALUNO$_v$ do esquema $\mathbf{S}_{v1}$ apresentado na Figura 5.3 e a assertiva de correspondência de extensão Ψ_2:ALUNO$_v$≡ESTUDANTE, a qual especifica a equivalência entre a classe de visão ALUNO$_v$ e a classe ESTUDANTE (Figura 5.2). De acordo com a tabela 5.1, essa assertiva de correspondência é mapeada para a assertiva de correspondência ALUNO$_v$≡EST&EMP[“ESTUDANTE” ∈ **classes**]. É importante notar que as assertivas de correspondência entre ALUNO$_v$ e a classe auxiliar EST&EMP refletem exatamente as assertivas de correspondência entre ALUNO$_v$ e as classes dos bancos de dados locais.

A seguir será mostrado como definir as classes de visão originais a partir das classes auxiliares mais detalhadamente.

5.4.1 Classes de Visão Originais Virtuais

No caso da classe de visão original **V** ser virtual, esta pode ser definida com uma simples seleção baseada nas assertivas de correspondência entre **V** e **A** (vide tabela 5.1).

Considere, por exemplo, a classe de visão EMPREGADO_v, apresentada na Figura 5.3, e a classe auxiliar EST&EMP, apresentada na Figura 5.5. Com base na assertiva de correspondência Ψ_3: $\text{EMPREGADO}_v \equiv \text{EST\&EMP}[\text{“EMPREGADO”} \in \textbf{classes}]$, é possível determinar a extensão de EMPREGADO_v. Isso é efeito através de uma simples seleção dos objetos de EST&EMP que são membros de EMPREGADO, como pode ser observado na Figura 5.6.

---

virtual class EMPREGADO_v *from* EST&EMP *extension* empregados
virtual atribute
nome$_v$: STRING *has value* *self*.**nome**;
ramal$_v$: STRING *has value* *self*.**ramal**;
nmDepto$_v$: STRING *has value* *self*.**nomeDepto**;
hide atribute
telefone, **curso**, **dependentes**, **rg**, **salario**, **classes**
includes
(*select* **e** *from* **e** *in* EST&EMP *where* “EMPREGADO” *in* **e**.**classes**)

Figura 5.6: Definição da classe de visão EMPREGADO_v

Na Figura 5.6 é mostrada a definição da classe de visão EMPREGADO_v, a qual foi escrita com base na linguagem O_2*Views*[56] [3]. A sintaxe dos comandos apresentados nessa Figura é descrita brevemente a seguir:

- A cláusula “**virtual class**” cria a definição de uma classe de visão em um esquema derivado de outros esquemas (reais ou também de visões).
- A cláusula “**from**” identifica a classe raiz da classe de visão.
- Na cláusula “**extension**” é declarado o nome através do qual a extensão da classe de visão é acessada na visão.

[3] A linguagem O_2*Views* é uma extensão da linguagem nativa do modelo de objetos O_2[57].

- Na cláusula "**virtual atribute**" são declarados os atributos da classe de visão com seus respectivos tipos e a expressão que determina seus valores.

- Na cláusula "**hide atribute**" são declarados os atributos que estão presentes na definição da classe auxiliar mas que não estão disponíveis para a classe de visão.

- Na cláusula "**includes**" é colocada a expressão que determina como os objetos da classe auxiliar serão selecionados para formar a extensão da classe de visão. É importante notar que a avaliação dessa expressão resulta apenas na extração dos objetos existentes da classe auxiliar que satisfazem a expressão. Isso significa que as instâncias da classe de visão são identificadas pelos OIDs dos objetos extraídos da classe auxiliar.

5.4.2 Classes de Visão Originais Materializadas

No caso da classe de visão **V** ser materializada, é criada uma extensão para **V** explicitamente, isto é, as instâncias da classe de visão são armazenadas fisicamente no banco de dados orientado a objetos ao invés de serem computadas quando requisitadas. É importante notar que, nesse caso, não é necessária a replicação de objetos, sendo que a extensão da classe de visão original conterá apenas as referências aos objetos da classe auxiliar que fazem parte da classe de visão. Isso só é possível devido aos seguintes fatos:

- Ao suporte a OIDs no modelo de objetos, pois no modelo relacional, sendo baseado em valores, os dados devem ser replicados para a visão.

- As classes auxiliares e as classes de visão estarem em um mesmo banco de dados.

- A existência de uma correspondência de 1-1 entre os objetos da classe auxiliar e de suas classes de visão.

A vantagem em armazenar apenas as referências dos objetos das classes auxiliares nas classes de visão originais é que os custos com a manutenção das visões materializadas são claramente reduzidos. Isso ocorre porque é necessário um menor espaço para armazenar as classes de visão, bem como é requerido um tempo menor para fazer a manutenção das mesmas. Para utilizar tal característica, é necessário existir um mecanismo para restringir a visibilidade dos objetos da classe de visão, ou seja, deve existir um modo de ocultar, nos objetos de uma classe de visão, as propriedades e métodos da classe auxiliar que não fazem parte da classe de visão. Até onde se sabe, esse mecanismo ainda não foi apropriadamente desenvolvido para visões materializadas (somente para visões virtuais)[4].

[4]Em [58], apenas as referências dos objetos das classes base são armazenados na classe de visão. No entanto, os autores não mencionam como a visibilidade desses objetos é restringida.

Outro problema surgido ao se materializar uma classe de visão **V** é que sua extensão deve ser atualizada quando a classe auxiliar **A** correspondente for modificada, a fim de que **V** fique consistente com as fontes de informação locais. Essa manutenção de **V** é feita diretamente a partir de **A** e sem acessar as fontes de informação locais. No enfoque proposto, as atualizações dos bancos de dados locais são propagadas para as classes auxiliares e as atualizações dessas classes são propagadas para as demais classes de visão. Isso é possível pois as assertivas de correspondência de **V** com os bancos de dados locais são deriváveis das assertivas de correspondência de **A** com os bancos de dados locais e das assertivas de correspondência de **A** com **V**. Portanto, pode-se garantir que se **A** está consistente com as fontes de informação locais e **V** está consistente com **A** então **V** está consistente com as fontes de informação.

É importante notar que a extensão de **V** é povoada uma primeira vez através da seleção dos objetos de **V** e depois é mantida incrementalmente. Para a manutenção incremental de **V**, devem ser geradas regras que propaguem uma atualização de **A** em atualizações de **V** de forma a tornar **V** consistente com **A**. Como são armazenadas apenas referências aos objetos de **A** na classe de visão **V**, a manutenção de **V** resume-se a inserções e remoções de objetos, isto é, as mudanças nos valores das propriedades de **A** não precisam ser tratadas.

As regras geradas para a manutenção da classe de visão **V** são geradas com base nas assertivas de correspondência de extensão entre **V** e **A**, definidas na tabela 5.1, e são descritas a seguir.

5.5 Gerando as Regras para a Manutenção da Classe de Visão V a partir da Classe Auxiliar A

Nesta seção considere as classes **V** e **A** como sugeridas na seção 5.4 e **a** um objeto de **A**. Para definir as regras para a manutenção de uma ACE de **V**, é necessário identificar as operações em **A** que causarão uma remoção ou inserção de objetos em **V**, as quais são:

- inserções em **A**.**classes**.
- remoções em **A**.**classes**.
- modificações dos valores das propriedades de **A** que compõem o predicado da classe de visão de seleção **V**.

A seguir serão apresentadas as regras que devem ser geradas para as assertivas de correspondência de extensão que caracterizam os vários tipos de classes de visão.

5.5.1 Manutenção das ACEs de Equivalência

Considere a ACE Ψ:$\mathbf{V} \equiv \mathbf{C}$, onde **V** é uma classe de visão de equivalência e **C** é uma classe raiz de **V**. De acordo com a tabela 5.1, a ACE Ψ é mapeada para a ACE Ψ': $\mathbf{V} \equiv \mathbf{A}[\text{"}\mathbf{C}\text{"} \in \mathbf{classes}]$.

A Figura 5.7 mostra a regra que deve ser gerada para fazer a manutenção da ACE Ψ' com relação às remoções em **a.classes**. A ação dessa regra consiste em remover o objeto **a** de **V** caso esse objeto fosse membro de **C**. Na Figura 5.8 é mostrada a regra que deve ser gerada com relação às inserções em **a.classes**, cuja ação consiste em inserir o objeto **a** em **V** caso esse objeto seja membro de **C**.

Quando remoção(v, **a**) em **A.classes** /* v é o nome da classe removido de **a.classes** */
Então Se v== "**C**" então
    Remova **a** de **V**

Figura 5.7: Regra para manutenção de Ψ': $\mathbf{V}\equiv\mathbf{A}[\text{"}\mathbf{C}\text{"} \in \mathbf{classes}]$ nas remoções em **a.classes**

Quando inserção(v, **a**) em **A.classes** /* v é o nome da classe inserido em **a.classes** */
Então Se v== "**C**" então
    Insira **a** em **V**

Figura 5.8: Regra para manutenção de Ψ':$\mathbf{V}\equiv\mathbf{A}[\text{"}\mathbf{C}\text{"} \in \mathbf{classes}]$ nas inserções em **a.classes**

5.5.2 Manutenção das ACEs de Diferença

Considere a ACE Ψ:$\mathbf{V} \equiv \mathbf{C}_1$ - $\mathbf{C}_2$, onde **V** é uma classe de visão de diferença, $\mathbf{C}_1$ e $\mathbf{C}_2$ são classes raízes de **V**. De acordo com a tabela 5.1, a ACE Ψ é mapeada para a ACE Ψ': $\mathbf{V} \equiv \mathbf{A}[\text{"}\mathbf{C}_1\text{"} \in \mathbf{classes} \land \text{"}\mathbf{C}_2\text{"} \notin \mathbf{classes}]$.

A Figura 5.9 mostra a regra que deve ser gerada para fazer a manutenção da ACE Ψ' com relação às remoções em **a.classes**. A ação dessa regra consiste em uma das seguintes atualizações:

- remover o objeto **a** de **V** caso esse objeto fosse membro de $\mathbf{C}_1$ e não seja membro de $\mathbf{C}_2$ ou;

Quando remoção(v, **a**) em **A**.**classes** /* v é o nome da classe removido de **a.classes** */
Então Se v== "$\mathbf{C}_1$" *e* "$\mathbf{C}_2$" $\notin$ **a.classes** então
  Remova **a** de **V**
 Senão
  Se v== "$\mathbf{C}_2$" *e* "$\mathbf{C}_1$" $\in$ **a.classes** então
   Insira **a** em **V**

Figura 5.9: Regra para manutenção de Ψ': $\mathbf{V}\equiv\mathbf{A}$["$\mathbf{C}_1$" $\in$ **classes** $\wedge$ "$\mathbf{C}_2$" $\notin$ **classes**] nas remoções em **a.classes**

- inserir o objeto **a** em **V** caso esse objeto seja membro de $\mathbf{C}_1$ e fosse membro de $\mathbf{C}_2$.

A Figura 5.10 mostra a regra que deve ser gerada para fazer a manutenção da ACE Ψ' com relação às inserções em **a.classes**. A ação dessa regra consiste em uma das seguintes atualizações:

- inserir o objeto **a** em **V** caso esse objeto seja membro $\mathbf{C}_1$ e não seja membro de $\mathbf{C}_2$ ou;

- remover o objeto **a** de **V** caso esse objeto seja membro de $\mathbf{C}_1$ e de $\mathbf{C}_2$.

Quando inserção(v, **a**) em **A**.**classes** /* v é o nome da classe inserido em **a.classes** */
Então Se v== "$\mathbf{C}_1$" *e* "$\mathbf{C}_2$" $\notin$ **a.classes** então
  Insira **a** em **V**
 Senão
  Se v== "$\mathbf{C}_2$" *e* "$\mathbf{C}_1$" $\in$ **a.classes** então
   Remova **a** de **V**

Figura 5.10: Regra para manutenção de Ψ': $\mathbf{V}\equiv\mathbf{A}$["$\mathbf{C}_1$" $\in$ **classes** $\wedge$ "$\mathbf{C}_2$" $\notin$ **classes**] nas inserções em **a.classes**

5.5.3 Manutenção das ACEs de União

Considere a ACE Ψ:$\mathbf{V} \equiv \cup_{i=1}^{m} \mathbf{C}_i$, onde **V** é uma classe de visão de união e $\mathbf{C}_1$, $\mathbf{C}_2$, ..., $\mathbf{C}_m$ são classes raízes de **V**. De acordo com a tabela 5.1, a ACE Ψ é mapeada para a ACE Ψ': $\mathbf{V} \equiv \mathbf{A}$["$\mathbf{C}_1$" $\in$ **classes** $\vee$ ... $\vee$ "$\mathbf{C}_m$" $\in$ **classes**].

A Figura 5.11 mostra a regra que deve ser gerada para fazer a manutenção da ACE Ψ' com relação às remoções em **a.classes**. A ação dessa regra consiste em remover o objeto **a** de **V** caso a classe removida em **a.classes** fosse a única responsável por tornar **a** membro de **V**. Na Figura 5.12 é mostrada a regra que deve ser gerada com relação às inserções em **a.classes**, cuja ação consiste em inserir o objeto **a** em **V** caso a classe inserida em **a.classes** torne o objeto **a** membro de **V**.

Quando remoção(v, **a**) em **A.classes** /* v é o nome da classe removido de **a.classes** */
Então Se "$\mathbf{C}_k$" $\notin$ **a.classes**, para k=1,m e $\mathbf{C}_k \neq v$ então
Remova **a** de **V**

Figura 5.11: Regra para manutenção de Ψ':$\mathbf{V}\equiv\mathbf{A}$["$\mathbf{C}_1$" $\in$ **classes** $\vee$ ... $\vee$ "$\mathbf{C}_m$" $\in$ **classes**] nas remoções em **a.classes**

Quando inserção(v, **a**) em **A.classes** /* v é o nome da classe inserido em **a.classes** */
Então Se "$\mathbf{C}_k$" $\notin$ **a.classes**, para k=1,m e $\mathbf{C}_k \neq v$ então
Insira **a** em **V**

Figura 5.12: Regra para manutenção de Ψ':$\mathbf{V}\equiv\mathbf{A}$["$\mathbf{C}_1$" $\in$ **classes** $\vee$ ... $\vee$ "$\mathbf{C}_m$" $\in$ **classes**] nas inserções em **a.classes**

5.5.4 Manutenção das ACEs de Interseção

Considere a ACE Ψ:$\mathbf{V} \equiv \cap_{i=1}^{m} \mathbf{C}_i$, onde **V** é uma classe de visão de interseção e $\mathbf{C}_1$, $\mathbf{C}_2$, ..., $\mathbf{C}_m$ são classes raízes de **V**. De acordo com a tabela 5.1, a ACE Ψ é mapeada para a ACE Ψ': $\mathbf{V} \equiv \mathbf{A}$["$\mathbf{C}_1$" $\in$ **classes** $\wedge$ ... $\wedge$ "$\mathbf{C}_m$" $\in$ **classes**].

Quando remoção(v, **a**) em **A.classes** /* v é o nome da classe removido de **a.classes** */
Então Se v== "$\mathbf{C}_i$" *e* "$\mathbf{C}_k$" $\in$ **a.classes**, para k=1,m e k $\neq$ i, então
Remova **a** de **V**

Figura 5.13: Regra para manutenção de Ψ':$\mathbf{V}\equiv\mathbf{A}$["$\mathbf{C}_1$" $\in$ **classes** $\wedge$ ... $\wedge$ "$\mathbf{C}_m$" $\in$ **classes**] nas remoções em **a.classes**

A Figura 5.13 mostra a regra que deve ser gerada para fazer a manutenção da ACE Ψ' com relação às remoções em **a.classes**. A ação dessa regra consiste em remover o objeto

a de **V** caso o objeto **a** fosse membro de todas as classes $\mathbf{C}_k$, para $1 \leq k \leq m$. Na Figura 5.14 é mostrada a regra que deve ser gerada com relação às inserções em **a.classes**, cuja ação consiste em inserir o objeto **a** em **V** caso esse objeto seja membro de todas as classes $\mathbf{C}_k$ (k=1,m).

Quando inserção(v, **a**) em **A.classes**   /* v é o nome da classe inserido em **a.classes** */
Então Se v== "$\mathbf{C}_i$" *e* "$\mathbf{C}_k$" $\in$ **a.classes**, para k=1,m e k $\neq$ i, então
    Insira **a** em **V**

Figura 5.14: Regra para manutenção de Ψ':**V**≡**A**["$\mathbf{C}_1$" $\in$ **classes** $\wedge$ ... $\wedge$ "$\mathbf{C}_m$" $\in$ **classes**] nas inserções em **a.classes**

5.5.5 Manutenção das ACEs de Seleção

Considere a ACE Ψ:**V** $\equiv$ **C**[**p**], onde **V** é uma classe de visão de seleção, **C** é uma classe raiz de **V** e **p** é um predicado de seleção. De acordo com a tabela 5.1, a ACE Ψ é mapeada para a ACE Ψ': **V** $\equiv$ **A**[**p** $\wedge$ "**C**" $\in$ **classes**].

Quando remoção(v, **a**) em **A.classes**   /* v é o nome da classe removido de **a.classes** */
Então Se v== "**C**" *e* **p**==*true* então
    Remova **a** de **V**

Figura 5.15: Regra para manutenção de Ψ':**V**≡**A**[**p** $\wedge$ "**C**" $\in$ **classes**] nas remoções em **a.classes**

A Figura 5.15 mostra a regra que deve ser gerada para fazer a manutenção da ACE Ψ: **V**≡**A**[**p** $\wedge$ "**C**" $\in$ **classes**] com relação às remoções em **a.classes**. A ação dessa regra consiste em remover o objeto **a** de **V** caso esse objeto fosse membro de **C** e o predicado **p** seja satisfeito. Na Figura 5.16, é mostrada a regra que deve ser gerada com relação às inserções em **a.classes**. A ação dessa regra consiste em inserir o objeto **a** de **V** caso esse objeto seja membro de **C** e o predicado **p** seja satisfeito.

Como mencionado no início do capítulo, no caso de uma classe de visão de seleção, as modificações do valor de uma propriedade **p** de **A** que compõe o predicado **p** de seleção são operações relevantes para a ACE Ψ'. Nesse caso, uma regra deve ser gerada para cada propriedade que compõe o predicado a fim de fazer a manutenção de Ψ'.

Quando inserção(v, **a**) em **A.classes** /* v é o nome da classe inserido em **a.classes** */
Então Se v== "**C**" *e* **p**==*true* então
Insira **a** em **V**

Figura 5.16: Regra para manutenção de Ψ':**V**≡**A**[**p** ∧ "**C**" ∈ **classes**] nas inserções em **a.classes**

Quando modificação(v, **a**) em **A**.***p*** /* v é o valor atribuído a **a**.***p*** */
Então Se "**C**" ∈ **a.classes** então
Se **antigo.p**== "*false*" *e* **p**== "*true*" então
Insira o objeto **a** em **V**
Senão
Se **antigo.p**== "*true*" *e* **p**== "*false*" então
Remova o objeto **a** de **V**

Figura 5.17: Regra para manutenção da ACE Ψ':**V**≡**A**[**p** ∧ "**C**" ∈ **classes**] nas modificações em **a**.***p***

A Figura 5.17 mostra a regra que deve ser gerada para a manutenção da ACE Ψ' nas modificações em **a**.***p***, onde ***p*** é uma propriedade monovalorada. A ação dessa regra consiste primeiramente em verificar se o objeto **a** é membro de **C**. Em caso afirmativo, uma das duas situações seguintes será executada:

- inserir o objeto **a** em **V** caso a seguinte situação tenha ocorrido: antes da modificação de **a**.***p***, o predicado de seleção **p** não era satisfeito no objeto **a**, porém após a modificação de **a**.***p***, o predicado **p** tornou-se verdadeiro[5] ou;
- remover o objeto **a** de **V** caso a seguinte situação tenha ocorrido: antes da modificação de **a**.***p***, o predicado de seleção **p** era satisfeito no objeto **a**, porém após a modificação de **a**.***p***, o predicado **p** tornou-se falso.

[5]Neste trabalho, é utilizada a notação **antigo.p** para significar o uso do predicado **p** com os valores do objeto **a** que antecede a modificação do valor de ***p***. Em um sistema como o Oracle8, o valor de uma propriedade ***p*** antes de uma modificação em **C**.***p*** é referenciado como "*old.p*"

Capítulo 6

Conclusões

Neste trabalho, foi abordado o problema da manutenção incremental e a auto-manutenção de classes de fusão em visões de integração de dados. Para solucionar esse problema, foi proposta uma estratégia para realizar a auto-manutenção de classes de fusão completas e foi mostrado como essas classes podem ser utilizadas como classes auxiliares para tornar auto-manuteníveis outros tipos de classes de fusão mais genéricos.

A manutenção das classes de fusão completas é feita através de regras locais, as quais propagam para o sistema global as atualizações a serem realizadas em uma classe de fusão completa. Uma inovação dessas regras locais com relação as regras utilizadas em outros enfoques [13] é que, todas as informações adicionais necessárias para fazer a manutenção da classe de fusão completa são determinadas localmente pelas regras locais e enviadas para o sistema global. Como mostrado nos exemplos apresentados, este enfoque resolve as limitações de [15], permitindo realizar a auto-manutenção de classes de fusão completas com propriedades derivadas e relacionamentos.

Outra inovação do enfoque proposto é que as regras locais são definidas a partir das assertivas de correspondência, as quais especificam formalmente o relacionamento entre o esquema da visão e os esquemas locais. Uma vantagem do uso das assertivas de correspondência é que o processo de geração das regras pode ser automatizado. Neste trabalho, foram desenvolvidos algoritmos que recebem como entrada as ACs de uma classe de fusão completa e geram automaticamente as regras necessárias para a sua manutenção. Outra vantagem do uso de assertivas de correspondência para a geração das regras, é que é possível provar formalmente que as regras geradas fazem corretamente a manutenção da visão. Isso significa que as atualizações solicitadas pelas regras locais para a visão refletem exatamente o efeito necessário para que a visão fique consistente com os bancos de dados locais.

As assertivas de correspondência também foram utilizadas para desenvolver os algo-

ritmos "A_1" e "A_2", apresentados nos apêndices A.1 e A.2 respectivamente. O algoritmo "A_1" é usado para gerar a operação "*GereEO_de_***F**", onde **F** é uma classe de fusão completa, enquanto que o algoritmo "A_2" é usado para gerar a operação "*GereEIO_de_***F**".

O enfoque proposto para a auto-manutenção de classes de fusão utilizando as classes de fusão completas consiste em:

- Utilizar as classes de fusão completas como classes auxiliares para permitir a auto-manutenção de classes de visão preservadoras de objetos.

- Definir as classes de visão originais usando apenas as classes auxiliares. Uma vez que as classes auxiliares são materializadas, as classes de visão originais podem ser virtuais ou materializadas. Isso representa uma vantagem do enfoque proposto pois permite uma maior flexibilidade para o usuário em decidir como implementar a classe de visão original.

No caso das classes de visão originais serem materializadas, então são criadas extensões para estas classes explicitamente. No presente trabalho, apenas as referências aos objetos da classe auxiliar que fazem parte da classe de visão original devem ser armazenadas, semelhante ao que ocorre em [58]. Assim sendo, a manutenção de uma classe de visão original resume-se a inserções e remoções de objetos, não sendo preciso propagar as mudanças dos valores das propriedades da classe auxiliar.

No presente trabalho, foi mostrado como as classes de visão originais virtuais podem ser definidas usando somente as classes auxiliares. Além disso, foram apresentadas todas as regras que devem ser geradas para fazer a manutenção das classes de visão originais materializadas. Tanto a definição da classe de visão virtual como a geração das regras para fazer a manutenção das classes de visão materializadas foram obtidas com base nas assertivas de correspondência, que especificam como as classes auxiliares estão relacionadas com as classes de visão originais.

Uma desvantagem dos enfoques que utilizam classes auxiliares para permitir a auto-manutenção de classes de visão é o enorme montante de dados adicional (as extensões das classes auxiliares) que deve ser armazenado no repositório de dados do sistema global e o custo de sua manutenção. Em sistemas de suporte à decisão, nem sempre essa é uma característica desejável, dado o grande volume de dados das próprias classes de visão originais. Como tópico para estudos futuros, é sugerido então o desenvolvimento de heurísticas, para determinar o conjunto mínimo de classes auxiliares que devem ser armazenadas no repositório de dados do sistema global de modo a não comprometer a eficiência do processo de manutenção. Note que nesse caso, é possível que alguma classe de fusão não possa ser auto-mantida.

Também é sugerido como trabalho futuro tratar:

- classes de visão geradoras de objetos, ou seja, aquelas classes em que não existem uma correspondência de um para um entre os objetos da classe de visão e da classe base.

- propriedades derivadas multivaloradas.

- propriedades computadas (classes de visão com agregações).

No futuro também pretende-se desenvolver uma ferramenta (vide Figura 6.1) para definir as classes de visão e para gerar as regras e demais operações necessárias para fazer a manutenção da visão materializada. A arquitetura da ferramenta mostrada na Figura 6.1 consiste dos seguintes módulos:

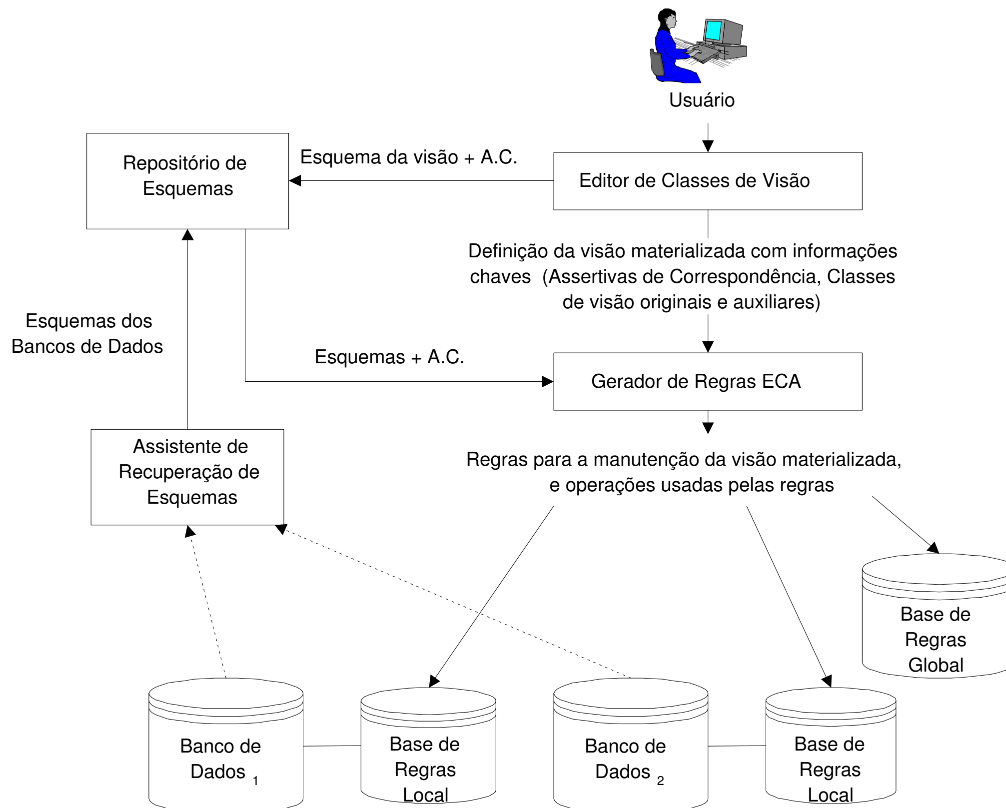

Figura 6.1: Ferramenta para a definição de classes de visão e das regras e operações para mantê-las

- *Editor de classes de visão*: onde o usuário define as classes de visão originais e as assertivas de correspondência entre essas classes e as classes raízes. Nesse módulo, são

geradas as classes auxiliares e as assertivas de correspondência entre as classes raízes e as classes auxiliares. Além disso, as assertivas de correspondência entre as classes originais e as classes raízes são mapeadas para as assertivas de correspondência entre as classes auxiliares e as classes de visão originais. Caso as classes de visão originais sejam virtuais, então estas são redefinidas usando somente as classes auxiliares.

- *Gerador de regras*: Gera as regras locais para a manutenção das classes auxiliares. Caso as classes de visão originais sejam materializadas, então também são geradas as regras globais para mantê-las.
- *Bases de regras locais*: armazenam as regras locais que fazem a manutenção das classes auxiliares.
- *Base de regras global*: armazena as regras globais que fazem a manutenção das classes de visão originais, caso estas sejam materializadas.
- *Repositório de esquemas*: armazena os esquemas externos dos bancos de dados locais, o esquema da visão e as assertivas de correspondência.
- *Assistente de recuperação de esquemas*: fornece as informações sobre a estrutura dos esquemas externos dos bancos de dados locais.

Bibliografia

[1] C. Batini, M. Lenzerini, S.B. Navathe, "A comparative analysis of methodologies for database schema integration," *ACM Computing Surveys*, vol. 18, pp. 323–364, Dezembro 1986.

[2] A.P. Sheth, J. Larson, "Federated database systems for managing distributed, heterogeneous and autonomous databases," *ACM Computing Surveys*, vol. 22, pp. 183–236, Setembro 1990.

[3] U. Dayal, H.Y. Hwang, "View definition and generalization for database integration in a multidatabase system," in *IEEE Trans. on Software Engineering*, vol. 10-No. 6, pp. 628–644, 1984.

[4] J M. Smith, P.A. Bernstein, N. Goodman, U. Dayal, T.A. Landers, K.W.T. Lin, E. Wong, "Multibase - integrating heterogeneous distributed database systems," in *National Computer Conference*, pp. 487–499, 1981.

[5] G. Wiederhold, "Mediators in the architecture of future information systems," in *IEEE Computer*, pp. 38–49, Março 1992.

[6] B. Lóscio, "Atualização de múltiplas bases de dados através de mediadores," Dissertação de Mestrado, Universidade Federal do Ceará, Fortaleza, Ceará, Brasil, 1998.

[7] J. Widom, "Research problems in data warehousing," in *Proceedings of the 4th International Conference on Information and knowledge Management(CIKM)*, (Baltimore, Maryland), pp. 25–30, Novembro 1995.

[8] S. Widjojo, R. Hull, D. Wile, "Distributed information sharing using worldbase," in *IEEE Office Knowledge Engineering*, pp. 17–26, Agosto 1989.

[9] Y. Zhuge, H.G. Molina, J. Hammer, J. Widom, "View maintenance in a warehousing environment," in *Proceedings of the ACM SIGMOD Symposium on the Management of Data*, (San Jose, California), pp. 316–327, Maio 1995.

[10] G. Zhou, R. Hull, R. King, "Squirrel phase 1: generating data integration mediators that use materialization," rel. téc., University of Colorado, Computer Science Department, Bounder, Novembro 1995.

[11] J.L. Wiener, H. Gupta, W. Labio, Y. Zhuge, H.G. Molina, J. Widom, "A system prototype for warehouse view maintenance," in *Proceedings of the ACM Workshop on Materialized Views: Techniques and Application*, (Montreal, Canada), pp. 26–33, Junho 1996.

[12] S. Ceri, J. Widom, "Deriving produtions rules for incremental view maintenance," in *Proceedings of the International Conference on Very Large Databases*, pp. 577–589, 1991.

[13] G. Zhou, R. Hull, R. King, "Generating data integration mediators that use materialization," *Journal of Intelligent Information Systems*, vol. 6(2/3), pp. 199–221, Maio 1996. Relatório Técnico CU-CS-793-95, University of Colorado, Computer Science Department.

[14] A. Gupta, H.V. Jagadish, I.S. Mumick, "Data integration using self-maintainable views," in *Proceedings of the 5th International Conference on Extending Database Technology (EDBT)*, (Avignon, France), pp. 140–144, Março 1996.

[15] A. Gupta, H.V. Jagadish, I.S. Mumick, "Maintenance and self-maintenance of outer-join views," in *The Third International Workshop on Next Generation Information Technologies and Systems*, (Neve Illan, Israel), 1997.

[16] A. Gupta, I.S. Mumick, ed., *Materialized views: techniques, implementations, and applications*, ch. Maintenance Policies, pp. 11–14. The MIT Press, 1997.

[17] W. Labio, H.G. Molina, "Efficient snapshot algorithms for data warehousing," in *Proceedings of the 22nd International Conference on Very Large Databases*, (Mumbai, Bambay, India), 1996.

[18] R. Hull, G. Zhou, "A framework for supporting data integration using the materialized and virtual approaches," in *SIGMOD record*, vol. 25(2), pp. 481–492, Junho 1996.

[19] H.G. Ribeiro, C. Collet, "Bahavior of active rules within multi-database systems," in *XIV Brazilian Symposium on Databases*, (Florianopolis), Outubro 1999.

[20] J. Faria, "Data-driven rules for the maintenance of derived data and integrity constraints in user interfaces to databases," rel. téc., Faculdade de Engenharia da Universidade do Porto, Porto, Portugal, 1999. INESC Relatório Técnico RI 1/99. Url: http://www.fe.up.pt/ jpf/research/tr3.ps.zip.

[21] H.G. Ribeiro, C. Collet, T. Coupaye, G. Vargas, "Towards an execution model for distributed active rules," *Troisièmes Journées Jeunes Doctorants en Systèmes d'information*, pp. 143–158, Março 1998.

[22] U. Dayal, E.N. Hanson, J. Widom, *Modern database systems: the object model, Interoperability, beyond.* Addison-Wesley, 1994.

[23] E.N. Hanson, J. Widom, "An overview of production rules in database systems," in *The kowledge Engineering Review*, vol. 8(2), pp. 121–143, Junho 1993.

[24] E. N. Hanson, "Rule condition testing and action execution in ariel," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Junho 1992.

[25] S. Potamianos, M. Stonebraker, "The postgres rules systems," in *Active Database Systems: Triggers and Rules For Advanced Database Processing 1996*, pp. 43–61, 1996.

[26] J. Widom, "The starburst rules system," in *Active Database Systems: Triggers and Rules For Advanced Database Processing 1996*, pp. 87–109, 1996.

[27] S. C. U. Dayal, A.P. Buchmann, "The hipac project," in *Active Database Systems: Triggers and Rules For Advanced Database Processing 1996*, pp. 177–206, 1996.

[28] N. Gehani, H.V. Jagadish, "Ode as an active database: constraits and triggers," in *Proceedings of the 17th International Conference on Very large Data Bases*, Setembro 1991.

[29] B. Adelberg, H.G. Molina, J. Widom, "The strip rule system for efficiently maintaining derived data," in *Procedings of the ACM SIGMOD International Conference on Management of Data*, (Tucson, Arizona, USA), Maio 1997.

[30] J. Dalrymple, *Extending rule mechanisms for the contruction of interoperable systems.* Tese de Doutorado, University of Colorado, Boulder, 1995.

[31] O. Boulcema, J. Dalrymple, M. Doherty, J-C. Franchitti, R. Hull, R. King, G. Zhou, "Incorporating active and multi-database-state services into an osa-compliant interoperability toolkit," in *The Collected Arcadia Papers*, 2nd ed., 1995.

[32] S. Ceri, C. Widom, "Production rules in parallel and distributed database environments," in *Proceedings of the 18th International Conference on Very Large Databases*, (Vancouver, Canada), pp. 339–351, 1992.

[33] A. Koschel, R. Kramer, G. von Bultzingsloewen, T. Belibel, P. Krumlinde, S. Schmuck, C. Weinand, "Configurable active functionality for corba," in *11th ECOOP'97 Workshop 7(CORBA: Implementation, Use, and Evaluation)*, (Finland), Junho 1997.

[34] J. Mylopoulos, A. Gal, K. Kontogiannis, M. Stanley, "A generic integration architecture for cooperative information systems," in *Proceedings of the 1st IFCIS International Conference on Cooperative Information Systems (CoopIS'96)*, (Brussels, Belgium), pp. 208–217, Junho 1996. IEEE.

[35] E.N. Hanson, S. Khosla, "An introduction to the triggerman asynchronous trigger processor," rel. téc., CISE Department, University of Florida, United States, Abril 1997. Reletório Técnico TR-97-007.

[36] G. von Bultzingsloewen, A. Koschel, P.C. Lockemann, H.-D. Walter, *Active Rules in Database Systems*, ch. ECA functionality in a distributed environment. Srpinger-Verlag, 1999. capítulo 8.

[37] G. Zhou, R. Hull, R. King, J. Franchitti, "Using object matching and materialization to integrate heterogeneous databases," in *Proceedings of the Third International Conference on Cooperative Information Systems (COOPIS-95)*, (Viena, Austria), Maio 1995.

[38] J.A. Blakeley, P.A. Larson, F.W. Tompa, "Efficiently updating materialized views," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pp. 61–71, 1986.

[39] L. Colby, T. Griffin, L. Libkin, I. Mumick, H. Trickey, "Algorithms for deferred view maintenance," in *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, (Montreal, Quebec, Canada), pp. 469–480, Junho 1996.

[40] A. Gupta, I. Mumick, V. Subrahmanian, "Maintaining views incrementally," in *Proceedings of the ACM SIGMOD*, pp. 157–166, 1993.

[41] J. Harrison, S. Dietrich, "Maintenance of materialized views in a deductive database: An update propagation approach," in *Proceedings of the 1992 JICLSP Workshop Deductive Databases*, pp. 56–65, 1992.

[42] X. Qian, G. Wiederhold, "Incremental recomputation of active relational expressions," in *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 3(3), pp. 337–341, Setembro 1991.

[43] H.A. Kuno, E.A. Rudensteiner, "Using object-oriented principles to optimize update propagation to materialized views," in *IEEE International Conference on Data Engineering, (ICDE-12)*, Março 1996.

[44] A. Gupta, H.V. Jagadish, I.S. Mumick, "Data integration using self-maintainable views," rel. téc., AT&T Bell Laboratories, Novembro 1994.

[45] R.G.G. Cattell, D. Barry, D. Bartels, M. Berler, J. Eastman, S. Gamerman, D. Jordan, A. Spreinger, H. Strickland, D. Wade, *The object database standard ODMG 2.0.* Morgan Kaufmann Publishers, 1997.

[46] Y. Papakonstantinou, H. Garcia-Molina, J.D. Ullman, "Medmaker: A mediation system based on declarative specifications," in *Proceedings of the 12th International Conference on Data Engineering* (S. Y. Su, ed.), (New Orleans, Louisiana), pp. 132–141, IEEE Computer Society, Fevereiro 1996.

[47] R. Krishnamurthy, W. Litwin, W. Kent, "Language features for interoperability of databases with schematic discrepancies," in *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data* (J. Clifford, R. King, ed.), (Denver, Colorado), pp. 40–49, ACM Press, Maio 1991.

[48] E. Rudensteiner, "Multiview: A methodology for supporting multiple views in object-oriented databases," in *Proceedings of the 18th International Conference on Very Large Databases*, pp. 187–198, 1992.

[49] Y. Papakonstantinou, S. Abiteboul, H.G. Molina, "Object fusion in mediator systems," in *Proceedings of the 22nd VLDB Conference*, (Mumbai, Bombay, India), 1996.

[50] S. Widjojo, R. Hull, D. Wile, "A specificational approach to merging persistent object bases," in *Implementing Persistent Object Bases* (Al Dearle, Gail Sahw, Stanley Zdonik, ed.), Morgan Kaufmann, Dezembro 1990.

[51] Y. Arens, C.Y. Chee, C.N. Hsu, C.A. Knoblock, "Retrieving and integration data from multiple information sources.," *International Journal of Intelligent and Cooperative Information Systems*, vol. 2(2), pp. 127–158, 1993.

[52] W. Kent, R. Ahmed, J. Albert, M. Ketabchi, “Object identification in multi-database systems,” in *Interoperable Database Systems (DS-5) (A-25)* (D. Hsiao, E. Neuhold, R. Sacks-Davis, ed.), North-Holland: Elsevier Science Publishers B. V., 1993.

[53] M.H. Scholl, H.J. Schek, M. Tresch, “Object algebra and views for multi-objectbases,” in *Distributed Object Management* (M. Tamer Özsu, Umeshwar Dayal, Patrick Valduriez, ed.), pp. 353–374, California: Morgan Kaufmann Publishers, 1994. In ”Workshop on Distributed Object Management”, Canada, Agosto, 1992.

[54] V.M.P. Vidal, M. Winslett, “Preserving update semantics in schema integration,” in *Proceedings of the Third International Conference on Information and Knowledge Management (CIKM'94)*, (Gaithersburg, Maryland-USA), pp. 263–271, Novembro 1994.

[55] G. Guerrini, E. Bertino, B. Catania, J.G. Molina, “A formal model of views for object-oriented database systems,” in *Theory and Practice of Object Systems*, vol. 3, pp. 157–183, 1997.

[56] C. dos Santos, “Design and implementation of an object-oriented view mechanism,” rel. téc., Institut National de Recherche en Informatique et en Automatique, France, Março 1994. Goodstep Esprit-III project No. 6115.

[57] F. Bancilhon, C. Delobel, P. Kannelakis, *Building an object-oriented database system - the story of O2.* Morgan Kaufmann, 1992.

[58] H.A. Kuno, E.A. Rudensteiner, “Materialized object-oriented views in multiview,” in *Proceedings RIDE-DOM '95, 5th International Workshop on Research Issues in Data Engineering - Distributed Object Management*, (Taipei, Taiwan), pp. 6–7, Março 1995.

Apêndice A

Algoritmos

A.1 Algoritmo $A_1()$

/* Gera a operação τ que obtém uma especificação de objeto $\mathbf{c}'$ da classe de fusão completa **F** que é semanticamente equivalente ao objeto **c** que foi inserido na classe raiz **C**. A especificação de objeto **c'** é dada por: ($\mathbf{p}_1$, ... , $\mathbf{p}_n$, **ClasseDeOrigem**), onde $\mathbf{p}_1, ..., \mathbf{p}_n$ são as propriedades de **F** e **ClasseDeOrigem** é o nome da classe **C**. */

{

01. $\tau = \emptyset$; /* Inicialização da operação τ */
02. $\tau = \tau \cup \{<$ inicializa as variáveis correspondentes às propriedades de **c'** com valores "null" ; $>\}$;
03. Para cada propriedade $\boldsymbol{p}$ da classe **C** faça
04. Se existir uma A.C. da forma $\mathbf{F}.\boldsymbol{p}_i \equiv \mathbf{C}.\boldsymbol{p}$, $1 \leq i \leq n$ então
05. Se $\boldsymbol{p}_i$ é um atributo então
06. $\tau = \tau \cup \{< v_{p_i} = \mathbf{this}.\mathbf{p};>\}$;
07. Senão /* $\boldsymbol{p}_i$ é um relacionamento */
08. Se **p** é monovalorado então
09. $\tau = \tau \cup \{< v_{p_i}$=**this**.**p**.*GereEIO_de_***D**;>}; /* *Dom*($\boldsymbol{p}_i$) =**D** */
10. Senão
11. $\tau = \tau \cup \{<$Para cada objeto **v** em **this**.**p** faça
12. v_{p_i}.*insert_element*(**v**.*GereEIO_de_***D**());>}; /* *Dom*($\boldsymbol{p}_i$) = **D** */
13. Se existir uma A.C. da forma $\mathbf{F}.\boldsymbol{p}_i \equiv \mathbf{C}.\boldsymbol{p} \bullet \boldsymbol{r'}_1 \bullet ... \bullet \boldsymbol{r'}_t$, onde $\boldsymbol{r'}_1$, ..., $\boldsymbol{r'}_t$ são propriedades das classes base $\mathbf{R}_1$, ..., $\mathbf{R}_t$, então
14. Se $\boldsymbol{p}_i$ é um atributo então
15. $\tau = \tau \cup \{< v_{p_i} = \mathbf{this}.\mathbf{p}.\boldsymbol{r'}_1. ... .\boldsymbol{r'}_t;>\}$;
16. Senão /* $\boldsymbol{p}_i$ é um relacionamento */
17. $\tau = \tau \cup \{<$v = **this**.**p**.$\boldsymbol{r'}_1$. ... .$\boldsymbol{r'}_t$;

18. $v_{p_i} = \text{v}.\mathit{GereEIO_de_}\mathbf{D}; >\}$; /* $Dom(\boldsymbol{p}_i) = \mathbf{D}$ */
19. Fim-se;
20. Fim-para;
21. $\tau = \tau \cup \{<\text{retorne}((\boldsymbol{p}_1{:}v_{p_1}, ..., \boldsymbol{p}_n{:}\ v_{p_n}, \textbf{ClasseDeOrigem}{:}\ \text{“C”}));>\}$;
22. retorne (τ);
}

A.2 Algoritmo $A_2()$

/* Gera a operação τ que obtém uma especificação de identificador de objeto id da classe de fusão completa **F** que é semanticamente equivalente ao objeto **c** da classe raiz **C**. A especificação de identificador de objeto id é dada por: $(\mathbf{p}_1, ..., \mathbf{p}_{n'})$, onde $\mathbf{p}_1, ..., \mathbf{p}_{n'}$ são propriedades de **F** usadas para determinar a equivalência semântica entre os objetos de **F** e **C** e cujos valores são obtidos a partir das propriedades de **c**. */

{
01. $\tau = \emptyset$; /* Inicialização da operação τ */
02. Para cada propriedade $\boldsymbol{p}$ da classe **C** tal que $\boldsymbol{p}$ é usada para determinar a equivalência semântica entre os objetos de **C** e **F** faça
03. Se existir uma A.C. da forma $\mathbf{F}.\boldsymbol{p}_i \equiv \mathbf{C}.\boldsymbol{p}$, $1 \leq i \leq n$ então
04. Se $\boldsymbol{p}_i$ é um atributo então
05. $\tau = \tau \cup \{< v_{p_i} = \textbf{this}.\mathbf{p}; >\}$;
06. Senão /* $\boldsymbol{p}_i$ é um relacionamento */
07. Se **p** é monovalorado então
08. $\tau = \tau \cup \{< v_{p_i} = \textbf{this}.\mathbf{p}.\mathit{GereEIO_de_}\mathbf{D}; >\}$; /* $Dom(\boldsymbol{p}_i) = \mathbf{D}$ */
09. Senão
10. $\tau = \tau \cup \{<$Para cada objeto **v** em **this.p** faça
11. $v_{p_i}.\mathit{insert_element}(\mathbf{v}.\mathit{GereEIO_de_}\mathbf{D}()); >\}$; /* $Dom(\boldsymbol{p}_i) = \mathbf{D}$ */
12. Se existir uma A.C. da forma $\mathbf{F}.\boldsymbol{p}_i \equiv \mathbf{C}.\boldsymbol{p} \bullet \boldsymbol{r'}_1 \bullet ... \bullet \boldsymbol{r'}_t$, onde $\boldsymbol{r'}_1, ..., \boldsymbol{r'}_t$ são propriedades das classes base $\mathbf{R}_1, ..., \mathbf{R}_t$, então
13. Se $\boldsymbol{p}_i$ é um atributo então
14. $\tau = \tau \cup \{< v_{p_i} = \textbf{this}.\mathbf{p}.\boldsymbol{r'}_1. ... .\boldsymbol{r'}_t; >\}$;
15. Senão /* $\boldsymbol{p}_i$ é um relacionamento */
16. $\tau = \tau \cup \{<\text{v} = \textbf{this}.\mathbf{p}.\boldsymbol{r'}_1. ... .\boldsymbol{r'}_t$;
17. $v_{p_i} = \text{v}.\mathit{GereEIO_de_}\mathbf{D}; >\}$; /* $Dom(\boldsymbol{p}_i) = \mathbf{D}$ */
18. Fim-se;
19. Fim-para;
20. $\tau = \tau \cup \{<\text{retorne}((\boldsymbol{p}_1{:}v_{p_1}, ..., \boldsymbol{p}_{n'}{:}\ v_{p_{n'}}));>\}$;
21. retorne (τ); }