MANUTENÇÃO DE RESTRIÇÕES DE INTEGRIDADE EM BANCOS DE DADOS ORIENTADOS A OBJETOS

Márcia Jacobina Brito Andrade

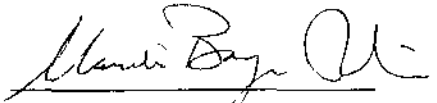
Prof. Dra. Claudia Bauzer Medeiros
Orientadora

Dissertação apresentada no Instituto de Matemática, Estatística e Ciência da Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

MANUTENÇÃO DE RESTRIÇÕES DE
INTEGRIDADE EM BANCOS DE DADOS
ORIENTADOS A OBJETOS

Este exemplar corresponde à redação final da tese devidamente corrigida e defendida pela Srta. Márcia Jacobina Brito Andrade e aprovada pela Comissão Julgadora.

Campinas, 25 de março de 1992

Prof. Dra. 
Claudia M. Bauzer Medeiros

Dissertação apresentada ao Instituto de Matemática, Estatística e Ciência da Computação, UNICAMP, como requisito parcial para obtenção do Título de MESTRE em Ciência da Computação.

"Tudo é uma questão de manter
a mente quieta,
a espinha ereta e
o coração tranqüilo"

Walter Franco

Aos meus pais e Mário,
que me ajudaram a alcançar um
sonho e, mesmo com toda a distância,
nunca deixaram de estar perto.

AGRADECIMENTOS

Gostaria de agradecer:

- Em primeiro lugar, a minha orientadora, Cláudia Bauzer Medeiros, pela constante assistência que recebi no decorrer da tese.
- Os comentários e sugestões dos professores que fizeram parte da banca, Prof. Dr. Geovane Cayres Magalhães e Prof. Dr. Léo Pini, bastante úteis para dar continuidade ao trabalho apresentado.
- Aos meus irmãos, Isabela e Dô, pelo incentivo que me deram.
- A Rackel, uma amiga sempre pronta a me ouvir e a me ajudar nos momentos difíceis. Levo com saudades lembranças das nossas conversas (madrugada adentro), risadas e "desesperos" (não podia faltar essa palavra nos meus agradecimentos!). Foi uma época inesquecível ...
- A Ju, pelas palavras de apoio que sempre ouvi.
- Aos meus amigos da Bahia, que não me deixaram esquecer o muito que eu tinha.
- Aos meus colegas e amigos de Campinas, por terem contribuído para as experiências que vivi nesses dois anos e que me fizeram olhar a vida de um modo diferente.

Sumário

Esta dissertação analisa o problema da manutenção de restrições de integridade estáticas em sistemas orientados a objetos, usando regras de produção e o paradigma de bancos de dados ativos. O trabalho mostra como transformar automaticamente restrições em regras de produção, a partir da restrição e do esquema do SGBD. O algoritmo de geração de regras foi implementado e pode ser usado não apenas por sistemas de bancos de dados orientados a objetos, mas também por sistemas relacionais e relacionais aninhados, sendo de uso geral. Além disso, como parte integrante do trabalho, foi desenvolvida uma taxonomia para restrições em sistemas OO, que considera sua dimensão dinâmica, e foi proposta uma linguagem de especificação de restrições para facilitar seu processamento. O trabalho estende propostas de outros autores, implementando suporte a restrições sobre dados e sobre métodos.

Abstract

This thesis analyzes the problem of static integrity constraints in object oriented database systems, using production rules and the paradigm of active databases. This work shows how to automatically change constraints into production rules, based on information from the constraints and the DBMS schema. The algorithm for rule generation was implemented, and can be used not only for constraint maintenance in object oriented database systems, but also for relational and nested database systems, being of general use. The research developed here includes the specification of a taxonomy for constraints in object oriented systems which considerates their dynamic dimension, and the definition and implementation of a language for constraint specification to facilitate their processing. This work extends proposals of other authors, implementing support for constraints, not only about data, but also about methods.

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

Esta dissertação aborda o problema da manutenção de restrições de integridade estáticas em sistemas orientados a objetos, usando regras de produção e o paradigma de banco de dados ativo.

O trabalho mostra como transformar automaticamente restrições estáticas em regras de produção, a partir da restrição e do esquema do SGBD. O algoritmo de geração de regras foi implementado e pode ser acoplado a vários tipos de sistemas de banco de dados. O algoritmo tem por objetivo detectar quais as operações efetuadas no banco de dados que devem ativar a verificação de uma determinada restrição, ou seja, quais as possíveis operações que podem levar à violação de uma restrição. Desse modo, busca-se atingir um melhor desempenho na manutenção da restrição, já que esta é verificada apenas a partir de um conjunto de operações que foram identificadas como relevantes.

É proposta também uma taxonomia de restrições de integridade para o modelo de dados orientado a objetos, visando definir as possíveis restrições neste modelo. Ademais, criou-se uma linguagem para especificar as restrições, a fim de facilitar seu processamento.

Este capítulo apresenta noções e terminologia básica que serão empregadas ao longo do texto.

1.1 Apresentação

O termo *integridade* [Dat89] é usado no contexto de banco de dados com o significado de correção, validade ou precisão. O problema da integridade é o de garantir que os dados no banco de dados sejam precisos, ou seja, o

problema de proteger o banco de dados contra atualizações inválidas, que podem ocorrer por diversos motivos: erros na entrada dos dados, erros do operador ou programador de aplicação, falhas do sistema e outros. Um outro termo utilizado no contexto de integridade é *consistência*.

Sabe-se que um banco de dados deve conter apenas dados "corretos", ou seja, deve manter a integridade dos dados, conforme já citado. Os critérios de correção dos valores especificados correspondem a um conjunto de condições às quais os valores devem obedecer. Entretanto, para que isso aconteça, dois tipos de controles costumam ser exercitados:

- Criação de uma estrutura ou projeto para o banco de dados capaz de permitir a representação de todos os estados possíveis que podem legitimamente ocorrer;
- Controle da entrada dos dados de forma a aceitar apenas os que representem valores consistentes e que correspondam à realidade.

O primeiro item é muitas vezes chamado "controle de correção por construção". Em outras palavras, procura-se garantir a correção a partir de uma modelagem adequada. Neste caso, a garantia de qualidade é restrita. Dependendo do tipo de modelo de dados usado na fase de projeto, quase não há restrições quanto aos possíveis estados definidos por projeto. É necessário, portanto, exercer outros controles durante modificações do banco de dados.

O segundo item é difícil de ser alcançado, pois exige controle operacional fora do sistema computadorizado: não apenas a consistência deve ser testada de acordo com as regras especificadas, mas é virtualmente impossível garantir que os dados consistentes reflitam a "realidade" do mundo exterior. Entretanto, se os dados puderem ser testados à entrada, pode-se evitar muitas ocorrências de dados que poderiam não representar fatos atuais. *Critérios de consistência* definem tais testes de *correção* dos dados, sendo verificáveis automaticamente.

As regras estabelecidas para que a consistência de um banco de dados seja alcançada são comumente denominadas *restrições de integridade* [JMSS90]. Assim, as restrições de integridade fornecem o meio de assegurar que as mudanças feitas ao banco de dados, por usuários autorizados, não levem à perda de consistência dos dados. Servem, deste modo, para proteger o banco de dados contra danos acidentais. Uma restrição de integridade é, então, uma condição que deve ser satisfeita sobre o banco de dados. Um pedido de atualização de um dado é aceito ou rejeitado, dependendo se as restrições de integridade são satisfeitas ou não.

Exemplos de restrições são as declarações de chaves no modelo relacional, que limita o conjunto de entradas e atualizações permissíveis e a forma de relacionamento no modelo E–R, onde o relacionamento um-para-um e um-para-muitos limita o conjunto de relacionamentos legais entre entidades de uma coleção de entidades.

De um modo geral, uma restrição de integridade pode ser um predicado arbitrário a ser testado sobre o banco de dados. Entretanto, testar predicados arbitrários pode ser de alto custo. Com isso, normalmente, as restrições de integridade consideradas são aquelas que podem ser testadas com um mínimo de custo adicional. Poucos sistemas permitem a expressão de restrições que sejam mais complexas que declarações de chaves ou restrições de domínio. Porém, é interessante que, ao se especificar um esquema de dados, possa ser incluído o conjunto de restrições sobre esses dados. Desse modo, um banco de dados, ao permitir a expressão de um maior número de restrições, facilita e diminui o trabalho do programador de aplicação.

Uma característica de um sistema de banco de dados seria, pois, garantir a manutenção automática de restrições. Esta característica deixaria os usuários completamente livres de preocupações sobre a preservação da consistência e, também, protegeria as informações armazenadas de operações incorretas.

A *visão conceitual* dos dados [Dat86] corresponde à representação abstrata do banco de dados em sua totalidade. Um *esquema conceitual* dos dados é uma definição desta visão. Sempre que for referenciado esquema de dados no texto, significa o esquema conceitual dos dados.

Este capítulo está organizado como se segue. A seção 2 discute conceitos de banco de dados orientado a objetos. A seção 3 apresenta uma visão geral de restrições em banco de dados. A seção 4 apresenta as restrições estáticas e discute as diversas maneiras de mantê-las. A seção 5 descreve sistemas ativos. E, finalmente, a seção 6 descreve sucintamente o trabalho de tese a ser apresentado.

1.2 Banco de Dados Orientado a Objetos

Inicialmente será mostrada a evolução dos bancos de dados, culminando com o advento dos Bancos de Dados Orientados a Objetos (BDOO). A seguir, serão apresentadas as áreas de aplicabilidade do modelo Orientado a Objetos (OO). E, por fim, serão descritas as características que fazem um banco de dados ser orientado a objetos.

1.2.1 Evolução dos Bancos de Dados

O modelo Relacional atingiu popularidade no início da década de 70 e, desde então, superou os modelos de dados hierárquico e de rede. As três abordagens (relacional, hierárquica e em rede) diferem na forma como elas permitem ao usuário ver e manipular relacionamentos. Na abordagem relacional, os relacionamentos são representados da mesma maneira que entidades, ou seja, tuplas em relações. Nas abordagens hierárquicas e em rede, certos relacionamentos são representados por meio de "interligações". O modelo relacional encontrou maior aceitação que o hierárquico e o de rede, pelo fato de representar melhor as entidades do mundo real através da organização dos dados em tabelas e de ser mais inteligível, além de ser o único com formalização teórica mais consolidada. Contudo, oferece dificuldade para modelagem semântica, o que motivou pesquisas por novos modelos de dados.

O primeiro modelo semântico bem-sucedido foi o modelo Entidade Relacionamento (ER) [Che76]. Não existem bancos de dados comerciais que sejam diretamente ER (em se tratando do ER conceitual), e a razão para isso é que aplicações modeladas com ER podem ser posteriormente convertidas em sistema relacional. É importante, contudo, ressaltar a preponderância deste modelo (desde o seu surgimento) para o projeto lógico de banco de dados, pois trouxe uma nova metodologia que foi e ainda é amplamente utilizada pelos administradores de banco de dados: os dados são modelados utilizando-se a metodologia ER e, posteriormente, mapeados para o modelo relacional.

Outra metodologia de modelagem importante é a do modelo Funcional, que enxerga um banco de dados como uma coleção de funções. Suponha a modelagem de Empregado, onde cada empregado tem um único identificador, no caso sua matrícula. Todos os atributos de um empregado podem ser considerados uma função da matrícula. Além disso, os relacionamentos entre entidades podem ser considerados uma função de um identificador de uma entidade para um outro identificador de outra entidade. Este modelo sofre do mesmo problema que o ER, por não representar uma mudança significativa em relação ao modelo relacional, além de ser difícil de implementar. Um exemplo de linguagem de acesso a um modelo funcional é a linguagem de dados DAPLEX [Shi81]. Nesta linguagem, os dados são formulados em termos de entidades. Existe uma representação funcional para os relacionamentos entre os dados e também uma coleção de construtores de linguagem para expressar critérios de seleção de entidades. Ademais, é introduzida a

noção de relacionamentos de subtipos/supertipos entre as entidades.

No fim da década de 70, surgiram modelos de dados semânticos, que incluíam construtores para modelagem, de forma a possibilitar a descrição de mais informações "semânticas" sobre os dados. Estes modelos incluem, usualmente, hierarquia de classes e composição, trazendo, assim, novos conceitos, como o de classe e o de herança, além de classificação e de agregação [HK87].

A modelagem semântica exerceu uma forte influência histórica sobre o modelo de dados Orientado a Objetos, que é o modelo mais recente a aparecer e cujo interesse comercial é grande. Modelos de dados semânticos, via-de-regra, não levam em consideração operações definidas pelo usuário. A adição desta característica é que distingue modelos semânticos de modelos orientados a objetos e que dá potencial de extensibilidade ao último, permitindo modelar a dinâmica da aplicação. Um modelo de dados semântico de grande destaque é o SDM [HM81]. Ele foi projetado com o objetivo de capturar mais do significado de um ambiente de aplicação do que era possível nos modelos de dados relacionais. Uma especificação SDM descreve um banco de dados em termos das entidades que existem no ambiente da aplicação, as classificações e agrupamentos destas entidades e as interconexões estruturais entre elas.

As linguagens de programação Persistentes merecem ser mencionadas pelo fato de terem contribuído historicamente para o surgimento dos BDOOs. As linguagens de programação tradicionais provêm facilidades para a manipulação de dados cujo tempo de vida não se estende além da ativação do programa. Se os dados devem sobreviver após a ativação de um programa, então deve ser utilizado algum arquivo de E/S ou uma interface de um Sistema de Gerenciamento de Banco de Dados (SGBD). Logo, os dados ou são manipulados pelas linguagens de programação ou são manipulados pelo SGBD. O mapeamento entre estes dois tipos de dados, em geral, é feito pelo sistema de arquivo ou pelo SGBD e também, em parte, pelo usuário que escreve e inclui nos programas código de tradução explícito.

As linguagens de programação persistentes surgiram como uma tentativa de se eliminar as diferenças existentes entre o SGBD e as linguagens de programação. O rompimento das diferenças foi feito com a identificação das melhores estruturas de dados e com a utilização da propriedade de dados chamada *persistência*. Um dado é persistente se ele sobrevive e é acessível além do escopo do processo que o criou.

Em [ABCM83] é apresentado o conceito de linguagem de programação persistente, onde a persistência é identificada como uma propriedade orto-

gonal dos dados, independentemente dos tipos e do modo como estes dados são manipulados. Sendo assim, é utilizado o princípio de que todos os dados, a despeito de seu tipo, devem ter os mesmos direitos, tanto para persistência como para a não-persistência. GALILEO [ACO85] é uma linguagem de programação que adere ao conceito de persistência. É uma linguagem interativa fortemente tipada, projetada especificamente para suportar características do modelo de dados semântico (classificação, agregação e especialização), assim como os mecanismos de abstração das linguagens de programação modernas (tipo, tipos abstratos e modularização).

1.2.2 Áreas de Aplicabilidade de BDOOs

Assim como o surgimento de bancos de dados tradicionais foi determinado principalmente como uma resposta às necessidades das aplicações comerciais típicas, o surgimento de BDOO também foi, de acordo com [ZM90], uma resposta às necessidades das aplicações de dados volumosos, aos problemas enfrentados pelo suporte dado pela engenharia de software a estas aplicações e ao problema do *impedance mismatch*, que quase sempre surge durante o desenvolvimento de aplicações de banco de dados.

Em se tratando das **aplicações de uso intensivo de dados**, a disponibilidade das estações gráficas de alto desempenho tem aumentado o escop. e a complexidade de tais aplicações, por exemplo: CAD (*Computer Aided Design*), CASE (*Computer-Aided Software Engineering*) e OIS (*Office Information System*).

Os sistemas de software que trabalham com essas aplicações requerem uma quantidade considerável de dados persistentes. Entretanto, o nível de complexidade desses programas e dados tem crescido muito. Os bancos de dados tradicionais não estão preparados para dar o suporte necessário.

O que se tem buscado, então, é estender a tecnologia de banco de dados a fim de que seja possível capturar facilmente semântica de dados específica da aplicação. Além disso, é importante que esta extensibilidade proporcione mecanismos para o desenvolvimento incremental das estruturas do banco de dados. O desenvolvimento de BDOOs tem sido direcionado para estas necessidades.

Com relação ao **suporte dado pela engenharia de software a essas aplicações**, o problema se refere à produção de sistemas complexos e extensos, que também requerem grande número de dados. As ferramentas CAD são exemplos de tais aplicações.

Esta complexidade dos sistemas não se manifesta apenas nos programas

que manipulam os dados, mas nos próprios dados. Os BDOOs solucionam a complexidade, incluindo facilidades para gerenciar o processo de engenharia de software (por exemplo, abstração de dados e herança) e características para capturar mais diretamente algumas das interconexões e restrições de dados (por exemplo, propriedades, relacionamentos e objetos complexos).

O problema do **impedance mismatch** tem relação com a dificuldade de comunicação entre uma linguagem de consulta e uma linguagem de programação, durante o desenvolvimento de aplicações que fazem acesso a um banco de dados. Existem dois aspectos para esta dificuldade de comunicação: um é a diferença nos paradigmas de programação (por exemplo, a linguagem declarativa SQL e a linguagem imperativa PL1) e o outro é o descasamento dos sistemas de tipos, propiciando perda de informações na interface, caso a linguagem de programação não esteja habilitada para representar diretamente estruturas do banco de dados como relações. Ademais, como existem dois sistemas de tipos, não existe modo automático para verificação de tipos na aplicação como um todo.

As linguagens de programação de banco de dados solucionam o problema do *impedance mismatch*, tornando persistentes os tipos de dados de uma linguagem de propósito geral ou adicionando tipos do banco de dados – listas ou relações – para o sistema de tipo da linguagem. Ainda assim, o problema continua se o acesso aos dados for feito via outras linguagens.

Os BDOOs tentam amenizar esse problema estendendo a linguagem de manipulação de dados para que a maior parte do código das aplicações seja escrito nesta linguagem e que o mínimo possível deste código seja escrito em uma linguagem de propósito geral.

1.2.3 Definição de Banco de Dados Orientado a Objetos

A nova geração de banco de dados, BDOO, incorporou tecnologia de outros campos: engenharia de software, linguagem de programação e inteligência artificial.

Os bancos de dados, ao permitirem inclusão de código adicional, estão acoplando o conceito da engenharia de software, que propõe uma boa organização a ser alcançada via modularidade. Sendo assim, uma das principais vantagens de um BDOO é que ele suporta o processo de engenharia de software para aplicações complexas, que requerem grandes números de dados persistentes.

No caso das linguagens de programação, o BDOO, ao prover uma linguagem de manipulação de dados computacionalmente completa, inclui muito

da execução da aplicação no próprio banco de dados. Também utilizam potentes técnicas de representação do conhecimento da inteligência artificial (IA), como, por exemplo, classificação e hierarquia de especificação, delegação de comportamento e outros, bem como utilizam técnicas de implementação comuns em IA: ligação dinâmica de mensagens para métodos e "garbage collection" automático.

A área de pesquisa em BDOO é uma área ativa no presente momento. Embora não haja nenhuma especificação clara do modelo orientado a objetos, existe um certo consenso sobre quais características são esperadas em um sistema de banco de dados orientado a objetos.

TERMINOLOGIA OO

Um *objeto* é uma máquina abstrata que define um *protocolo* através do qual os seus usuários podem interagir. O objeto possui um estado que é armazenado na memória de forma encapsulada. Esta é uma característica fundamental das linguagens OO.

O protocolo de um objeto é definido por um conjunto de *mensagens* com *assinaturas* tipadas. Uma mensagem pode ser enviada para um objeto com o fim de executar alguma ação. Uma mensagem é implementada por um *método*. Um método é como o corpo de um procedimento e possui privilégios especiais que permite fazer acesso à representação da memória privada do objeto para o qual a mensagem foi enviada. Sendo assim, um método é caracterizado pela sua assinatura (seu nome, seu tipo e o tipo de seus parâmetros) e pelo seu corpo (procedimento).

Todo objeto é ocorrência de uma *classe*. Logo, classe é a forma utilizada para reunir objetos semelhantes. A classe define as mensagens para as quais os objetos devem responder e, também, define como os objetos desta classe são implementados.

As classes são representadas em um grafo direcionado com as arestas conectando as *superclasses* com as *subclasses*, que representam os relacionamentos IS-A. Uma subclasse herda o comportamento de suas superclasses, podendo adicionar novos comportamentos.

Abaixo são descritas as características para um banco de dados ser orientado a objetos, baseado em [ABD+89]:

- Deve ser um Sistema de Gerenciamento de Banco de Dados e como

tal incluir as características:

- persistência;
- gerenciamento de memória secundária;
- concorrência;
- recuperação de dados;
- facilidade de consulta "ad hoc".

- Deve suportar *identidade de objetos.*
- Deve prover *encapsulamento.* Este encapsulamento deve ser a base em que todos os objetos abstratos são definidos.
- Deve suportar *objetos complexos.*
- Deve permitir *sobreposição* e *acoplamento tardio* ("late binding").
- Deve possuir o conceito de *tipo ou classe* como mecanismo de descrição de objetos idênticos.
- Deve permitir *herança e hierarquia de tipos.*
- Deve prover *extensibilidade.*

Identidade de Objetos

Identidade é a propriedade de um objeto que distingue cada objeto de todos os outros [KC86]. Identidade tem sido pesquisada independentemente em linguagens de banco de dados e em linguagens de programação de propósito geral.

Muitas linguagens de programação e de banco de dados usam nomes de variáveis para distinguir objetos temporários, misturando endereçamento e identidade. Muitos sistemas de banco de dados usam chaves para distinguir objetos persistentes, misturando valores de dados e identidade. Outrossim, a identidade fica comprometida. Linguagens orientadas a objetos empregam mecanismos separados para estes conceitos, de modo que cada objeto mantém uma noção consistente e separada da identidade, ao invés de utilizar o acesso aos dados e seus valores como mecanismos de detecção da identidade do objeto.

Em um modelo com identidade de objetos, um objeto tem uma existência que é independente de seu valor. Os modelos de dados orientados a objetos

possuem a habilidade de fazer referências através da identidade do objeto. Nos modelos de dados baseados em valor, entidades são identificadas por um subconjunto de seus atributos, chamado de chave. A identidade do objeto, por sua vez, é imutável e visível apenas em nível do sistema.

Dessa maneira, surgem duas noções de equivalência de objetos: dois objetos podem ser idênticos (têm o mesmo identificador) ou podem ser iguais (possuem o mesmo valor). A identidade é refletida em geral no chamado "oid" (object identifier), que corresponde, a grosso modo, à chave genética de um objeto.

Uma questão freqüentemente levantada é se uma chave primária em um banco de dados relacional significa o mesmo que a identidade do objeto?!. A resposta é não. Chaves primárias são únicas apenas dentro de uma única relação. Identidades de objetos são únicas em todo o banco de dados e não podem ser reaproveitadas.

Apesar da chave não ser a identidade do objeto, ela é utilizada em BDOOs. Neste caso, serve para identificar unicamente um objeto dentro de uma coleção.

Encapsulamento

O conceito de encapsulamento, tal como considerado em mecanismo de objetos, surgiu em função de tipos abstratos de dados. Usando estes conceitos, um objeto é considerado como tendo dois componentes: interface e implementação. A parte de interface é a especificação do conjunto de operações que podem ser executadas sobre o objeto. É a parte do objeto visível a outros objetos. A parte de implementação possui dados e procedimentos. Os dados representam o objeto e os procedimentos descrevem a implementação de cada operação.

O comportamento dos objetos deve ser completamente caracterizado pelo seu conjunto de operações. Esta propriedade é garantida se as operações forem o único modo direto de criação e manipulação dos objetos. Um efeito desta restrição é que, ao se definir uma abstração, o programador deve ter o cuidado de incluir um conjunto suficiente de operações, já que toda ação que ele desejar executar deve ser realizada em termos deste conjunto.

Abstração de dados através do encapsulamento é uma das características chaves de BDOO. A linguagem de programação CLU [LSAS77] introduziu abstração em uma linguagem com verificação de tipo forte e estática. O conceito central em CLU é o *cluster*. O *cluster* é um mecanismo de abstração de dados que tem uma representação de tipo e uma interface definida por

um conjunto de operações. Apenas estas operações têm o privilégio de fazer acesso à representação dos objetos daquele tipo.

Um objeto encapsula tanto o programa quanto os dados. Por exemplo, considere a entidade *Empregado* como um objeto que possui dados e operações (por exemplo, aumentar-salário, demitir-funcionário). Ao armazenar o conjunto de Empregados, tanto os dados quanto as operações que dizem respeito a este conjunto, são armazenados no banco de dados. No entanto, os dados só podem ser manipulados através destas operações.

O encapsulamento provê uma forma de independência lógica dos dados: pode-se modificar a implementação de uma operação sem modificar qualquer programa que a utiliza.

Objetos Complexos

Chama-se objeto complexo aquele composto de outros objetos, sendo que as regras de composição permitem qualquer combinação utilizando um conjunto predeterminado de operadores (por exemplo, tuplas, conjuntos, listas e vetores). Assim, o estado de um objeto complexo contém valores atômicos (por exemplo, inteiros, caracteres) e estados de outros objetos.

Desse modo, é possível que mais de um objeto se referencie a um mesmo objeto. Esta capacidade dos objetos compartilharem componentes relacionados é útil, pois qualquer mudança no objeto compartilhado é refletida nos objetos que o contém.

Sobreposição e Acoplamento Tardio

Existem casos onde se quer ter o mesmo nome para diferentes operações. Considere, por exemplo, a operação "display", que mostra o conteúdo de um objeto na tela. Dependendo do tipo do objeto, usa-se diferentes mecanismos de "display". Define-se, então, esta operação na classe mais geral e também nas subclasses que necessitam de operações de "display" especiais. Tal redefinição das operações nas subclasses é chamada *sobreposição*.

A fim de permitir esta sobreposição de operações, é necessário que a ligação dos nomes das operações com seu código seja resolvida em tempo de execução (*acoplamento tardio*).

Herança e Hierarquia de tipos

Herança é um termo que descreve muitos mecanismos diferentes que permitem as definições ou implementações de tipos estarem relacionadas através

de uma ordem parcial. A noção básica é que é possível modificar as definições de tipos incrementalmente, adicionando definições de subtipos que modificam o tipo original. A definição do supertipo, aliada às modificações oriundas do subtipo, produzem um novo tipo. Por exemplo, imagine as entidades *Empregado* e *Estudante* como subclasses da classe *Pessoa*, por possuírem características embutidas nesta última. Entretanto, Empregado e Estudante possuem características distintas: Empregado possui a operação relacionada com o pagamento do salário e Estudante possui a operação de cálculo do seu coeficiente de rendimento. Estas duas subclasses herdam os atributos e as operações da classe *Pessoa*. Também, possuem componentes e operações próprias.

A introdução da herança nas linguagens orientadas a objetos compromete bastante os benefícios trazidos pela abstração de dados. Em [Sny86] é examinado o relacionamento entre herança e encapsulamento e são desenvolvidos requisitos para o total suporte do encapsulamento com herança.

A herança ajuda a reusabilidade do código. São destacados quatro tipos de herança, por substituição, inclusão, restrição e especialização:

- Substituição: uma classe **c** herda de uma classe **sc**, caso se possa executar mais operações sobre objetos da classe **c** que sobre objetos da classe **sc**. Este tipo de herança é baseado no comportamento e não em valores.

- Inclusão: corresponde à noção de classificação. Se todo objeto de **c** é também da classe **sc**, então **c** é subclasse de **sc**. É uma herança baseada na estrutura e não nas operações.

- Restrição: é um subcaso da herança de inclusão. Uma classe **c** é subclasse da classe **sc**, se **c** consiste de todos os objetos da classe **sc** que satisfaçam a uma dada restrição. Ex.: considere Jovem como subclasse da classe Pessoa. Jovem não possui mais campos ou operações, apenas obedece a restrição: idade entre 13 e 19 anos.

- Especialização: uma classe **c** é uma subclasse da classe **sc**, se objetos da classe **c** são objetos da classe **sc** e contêm mais informações específicas. Por exemplo, a classe Empregado como subclasse da classe Pessoa.

Extensibilidade

O sistema de banco de dados suporta um conjunto de tipos predefinidos, os quais podem ser usados por programadores para escrever suas aplicações.

Este conjunto de tipos deve ser extensível no seguinte sentido: existe um modo de definir novos tipos a partir dos tipos predefinidos e não existe distinção pelo sistema no uso de tipos definidos pelo sistema e definidos pelo usuário. A definição de tipos inclui a definição de operações sobre eles.

1.3 Restrições

Existem vários tipos de classificação de restrições de integridade. Uma delas considera o modelo relacional e diz respeito à localidade da restrição quanto a uma relação: restrições *inter-relação* e *intra-relação*. Restrição de inter-relação ocorre quando a restrição é estabelecida entre mais de uma relação, e.g., às vezes deseja-se garantir que um conjunto de valores de um atributo em R1 seja o mesmo conjunto de valores em uma outra relação R2 . Restrição de intra-relação ocorre quando a restrição é estabelecida dentro da própria relação, podendo ser horizontal (relacionando atributos de uma tupla) ou vertical (relacionando valores de um atributo). Outra classificação possível é a que distingue restrições *sobre estados* e *sobre procedimentos*. Restrição sobre estados ocorre quando a restrição é estabelecida sobre os dados, ou seja, o controle é feito sobre a entrada de dados. Restrição sobre procedimentos ocorre quando a restrição é estabelecida sobre procedimentos que atuam sobre os dados, em outras palavras, o controle é feito, por exemplo, sobre como/quando modificá-los, quem pode ativá-los, como/quando executá-los.

Em [JMSS90] são apontados diferentes critérios para diferenciar restrições de integridade: a localização dos dados sobre os quais são expressas (por exemplo, o campo de uma tupla, toda a tupla, uma relação, muitas relações); o escopo temporal; a complexidade dos algoritmos necessários para verificação da restrição; facilidade de manutenção das estruturas de dados; o instante em que as restrições devem ser verificadas (por exemplo, no fim da transação, em um determinado horário do tempo real ou quando determinadas operações são executadas); a espécie de ação tomada quando a restrição é violada (por exemplo, aborto de transação, mensagem de erro, medidas corretivas).

Neste trabalho, será feita distinção entre restrições *estáticas* e *dinâmicas*.

Restrições de Integridade Estáticas são predicados especificados sobre um estado do banco de dados. Se todos estes predicados são avaliados como verdadeiros para um dado estado, então o estado é consistente.

Restrições de Integridade Dinâmicas são predicados especificados sobre

uma seqüência de estados. Restrições dinâmicas são de difícil verificação. Em geral, tenta-se transformar cada restrição dinâmica em um conjunto de restrições estáticas, para permitir seu processamento. Algumas vezes, uma restrição é transformada em um grafo de transição de estados, onde os nodos terminais representam estados consistentes. Uma considerável quantidade de restrições dinâmicas pode ser tranformada em expressões que referenciam apenas estado inicial e final de uma *transação.*

Entende-se por transação uma seqüência atômica de ações, conduzindo de um estado inicial do banco de dados para um estado terminal. Uma transação é uma série de atualizações e acessos ao banco de dados cujas mudanças intermediárias são invisíveis aos seus usuários. Se uma transação termina com sucesso, todas as atualizações feitas tornam-se visíveis de um modo atômico, isto é, simultaneamente. Se uma transação falha por razões externas, razões de consistência (como violação de restrições) ou razões de programação (como exceções), o banco de dados é restaurado para o estado anterior à execução da transação.

Restrições dinâmicas que podem ser expressas em termos dos estados iniciais e finais de uma única transação são chamadas *restrições de predicados de dois estados* [JMSS90]. Por exemplo, a restrição "o salário de um empregado não pode ser reduzido" necessita de dois estados do salário do empregado, o anterior e o atual. Um tipo de restrição dinâmica especial é a chamada restrição *temporal*, que necessita da manutenção de informações históricas sobre o estado do banco de dados e de controle de eventos no tempo. Um exemplo é a restrição "um empregado que foi gerente três vezes, não pode ser demitido". As restrições temporais são excessivamente difíceis de serem mantidas, caso não se consiga reduzi-las para restrições estáticas ou de dois estados.

Restrições estáticas são usualmente expressas através de lógica de primeira ordem, e restrições dinâmicas, através de lógica temporal.

Restrições dinâmicas foram, até agora, pouco estudadas, devido à sua complexa manutenção. Alguns estudos teóricos foram feitos recentemente, mas sua aplicabilidade é ainda bastante restrita, no caso de restrições que não levem a predicados de dois estados [JMSS90, Kun85].

1.4 Restrições Estáticas

A restrição estática é também chamada restrição *baseada em estado*, já que atua sobre um determinado estado do banco de dados.

Exemplos de restrições estáticas são dependência funcional (FD) e dependência multivalorada (MVD), bastante comuns no modelo relacional. Dada uma relação R, o atributo Y de R é *funcionalmente dependente* do atributo X de R se e somente se, sempre que duas tuplas de R combinarem em seus valores de X, elas também combinem no valor de Y. Chaves representam um tipo especial de dependência funcional. Dada uma relação R com atributos A, B, C, a *dependência multivalorada*, R.A $\rightarrow\!\!\!\rightarrow$ R.B, vale para R se e somente se, o conjunto de valores B que se combinam com um dado par (valores de A, valores de C) em R depender somente do valor de A e for independente do valor de C (A, B, C podem ser compostos). Informalmente, uma dependência multivalorada de valores de A para B, C na relação R (A, B, C) significa que, para cada valor de A, existe produto cartesiano dos valores B e C correspondentes. Para um melhor entendimento, imagine a relação com os atributos curso, professor e texto, onde curso pode ser ministrado por qualquer dos professores indicados e usa todos os textos indicados. Um exemplo de uma MVD é curso $\rightarrow\!\!\!\rightarrow$ professor que significa intuitivamente que, embora um curso não tenha um único professor, cada curso tem um conjunto bem definido de professores correspondentes, ou seja, para um curso c e um texto t, o conjunto p de professores, que correspondem ao par (c, t), depende só de c. Além disso, todo professor de c tem acesso a todos os textos de c.

Restrições estáticas são, em geral, expressas em lógica de primeira ordem.

Alguns exemplos de restrições estáticas escritas em linguagem corrente são:

1. A matrícula de um empregado é menor que 20.000;
2. Um gerente tem um salário maior que cr$1.500.000,00;
3. O número da matrícula do empregado é único (integridade de chave);
4. A data-início de um projeto não pode ser maior que a data-fim dos projetos dos quais ele depende;
5. Um empregado deve pertencer a um dos departamentos da empresa (integridade referencial).

As restrições de integridade estáticas só expressam fatos que estão representados no banco de dados. Por exemplo, a restrição de que uma pessoa não pode ter o seu salário reduzido em relação a qualquer salário anterior, não pode ser expressa usando restrição de integridade baseada em estado, caso

o banco de dados mantenha registros apenas dos salários correntes. Entretanto, se o banco de dados incluir histórico do salário, então tal consistência pode ser garantida através de um predicado estático.

Abaixo serão mostradas as diversas maneiras de se manter restrições estáticas.

1.4.1 Maneiras de Manter Restrições Estáticas

Conforme mencionado na introdução, restrições podem ser mantidas através de dois tipos de mecanismos:

1. Embuti-las na construção (no projeto) do esquema. Dependências de chaves, e.g., são tipicamente especificadas nesta fase. Aplicando-se normalização dos dados, consegue-se facilidades para a manutenção de algumas restrições estáticas: restrições de chave, restrições referenciais e outras.

2. Controlá-las quando das operações de atualização. Neste caso, há dois tipos de tratamento possível, se a atualização violar alguma restrição:

   - Bloquear a atualização ou
   - Aceitar e propagar a atualização. A verificação da consistência pode ser imediata ou diferida.

Esta seção se ocupa do segundo tipo de manutenção de restrições (controlado).

Bloquear a atualização

Se uma determinada atualização violar uma restrição, esta é imediatamente bloqueada. Neste caso, o usuário é avisado através de uma mensagem. Este é o único tipo de controle suportado por sistemas comerciais, para alguns tipos de restrições estáticas.

Aceitar e propagar a atualização

Este tipo de controle é exercido por programa, independentemente do SGBD, podendo, o programa, estar embutido na própria aplicação ou consistir numa chamada adicional de verificação de restrições.

Verificação imediata significa que o banco de dados deve ser verificado para consistência assim que ocorre alguma atualização.

Verificação diferida significa que se pode querer diferir o teste de consistência para um outro momento. Este conceito está relacionado com o de transação, definido na seção 1.3. Uma transação é considerada uma unidade de programa atômica. Desse modo, durante a execução de uma transação, o banco de dados pode ficar inconsistente. Logo, é importante que a verificação da integridade seja diferida até que a transação chegue ao final.

A decisão de diferir depende de vários fatores: se as ocorrências correspondentes não são de interesse para a aplicação corrente, é possível diferir a verificação até que as ocorrências se tornem de interesse. A este tratamento dá-se o nome de Verificação de Integridade Diferida, oposto ao mais tradicional – Verificação de Integridade Imediata [Laf82].

Um argumento a favor da verificação de integridade diferida é que esta freqüentemente corresponde ao mundo real, já que tem como hipótese tolerância de violações temporárias de integridade.

É interessante ressaltar as desvantagens da verificação imediata e da diferida. Na primeira, um registro pode ser modificado muitas vezes, devido a muitas modificações em outros, antes que qualquer aplicação necessite dele. Entretanto, apenas a modificação mais recente é a que interessa. A segunda requer processamento posterior de atualização. Em [Laf82] as duas alternativas foram comparadas. A princípio, nenhuma das duas pareceu superior, mas, após simulações, verificou-se que a diferida possuía um custo menor, principalmente em se tratando de bancos de dados de muitas atualizações como os bancos de dados CAD.

Um exemplo de atualização diferida pode ser encontrado em [CFT88], onde é descrito um projeto de um monitor que garante integridade referencial, e as operações submetidas pelo usuário podem ser propagadas ou modificadas. A propagação é implementada executando novas operações quando a sessão termina, usando dados coletados durante o processamento.

Um outro exemplo de atualização diferida pode ser visto em [SkS86], onde é descrito um algoritmo que processa atualizações que chegam desordenadamente, de modo a manter uma visão consistente dos dados. As atualizações chegam fora de ordem, mas, associada a cada uma delas, existe um "timestamp", e os dados do banco de dados devem refletir as atualizações por ordem de "timestamp". Por causa de atrasos de comunicação, uma atualização que está chegando pode ter "timestamp" menor que uma outra que já chegou. Se as atualizações já feitas entram em conflito com as que estão chegando, elas devem ser desfeitas e reexecutadas para manter o desejado critério de ordenação por "timestamp". O algoritmo usa um "log" para a reexecução de determinadas atualizações. O seu objetivo é o de controlar

a quantidade de reexecuções necessárias e, também, dispender um esforço moderado de computação para determinar as atualizações que devem ser reexecutadas.

Em [Md86] é discutido um sistema de condensação e diferimento de atualizações em banco de dados relacional, que utiliza para o pré-processamento um "log" de tipo condensado, acessado através de índices em árvore-B. [Md86] difere de [CFT88] e [SkS86] pois, em [Md86], é proposta uma estrutura de dados para o suporte do diferimento de atualizações, acrescentando, desse modo, a idéia da implementação.

1.5 Sistemas Ativos

Sistemas de banco de dados têm, via-de-regra, suporte limitado à manutenção automática de restrição. Programadores de aplicação são forçados, portanto, a incluir em seus sistemas código que efetua a verificação das restrições, sobrecarregando, com isso, as aplicações e tornando-as dependentes de modificações nos dados.

Uma solução apresentada por vários pesquisadores é a de embutir no próprio SGBD um sistema de regras ou gatilhos, que é acionado quando há alguma atualização. Isto permite que a verificação de restrição se faça de forma independente das aplicações [MP91b].

Uma das derivações do conceito básico de regras é denominado em IA de *regras de produção*. As regras possuem a forma **Se** X, **então** Y, onde X é a condição e Y é a ação. O par <X,Y> é executado quando da ocorrência de um evento específico. Sistemas de Inteligência Artificial usam, por exemplo, regras de produção e objetos ativos. A introdução das regras em um SGBD possibilita que este se torne ativo.

Um banco de dados é dito *passivo* quando oferece pouco ou nenhum suporte para o gerenciamento automático de condições definidas sobre o estado do banco de dados em resposta a estímulos externos. Tradicionalmente, os SGBDs são passivos, ou seja, executam transações apenas quando explicitamente requisitadas pelo usuário ou programa de aplicação.

Um banco de dados é dito *ativo* quando eventos gerados interna ou externamente ao sistema provocam uma resposta do banco de dados, independentemente da solicitação especificada pelo usuário. Neste caso, alguma ação é tomada dependendo das condições que foram impostas sobre o estado do banco de dados. O paradigma de banco de dados ativo é útil para implementar muitas funções do próprio banco de dados ou para estender estas

funções. Alguns exemplos são: controle de integridade, controle de acesso, manuseio de dados derivados e definição e aplicação de mecanismos de herança em modelo de dados orientado a objetos. Em [Mor83], o termo "banco de dados ativo" surge para descrever um sistema que suporta atualizações automáticas de visões e dados derivados à medida que os dados originais são atualizados.

1.6 Organização da Tese

A tese está organizada como se segue. O capítulo 2 discute sistemas ativos. O capítulo 3 apresenta uma taxonomia de restrições para bancos de dados orientados a objetos (BDOO). O capítulo 4 descreve o algoritmo para transformação de restrições em regras e sua utilização em sistemas OO e relacionais. O capítulo 5 apresenta a implementação desse algoritmo visando o sistema O_2 [Deu90]. O capítulo 6 contém a conclusão e algumas sugestões para extensão deste trabalho. O anexo contém o diagrama sintático da linguagem de restrição criada e o pseudocódigo dos módulos implementados.

Capítulo 2

Revisão Bibliográfica

2.1 Tratamento de Restrições

Há muitas propostas para a garantia de restrições de integridade – gatilhos e asserções, regras de integridade e procedimentos engatilhados e outros. Sistemas de Inteligência Artificial usam, por exemplo, regras de produção e objetos ativos. Todavia, segundo [DBM88], apenas os SGBDs ativos conseguem suportar e solucionar problemas mais complexos, posto que provêem capacidade para estruturar e recuperar regras e fatos, permitem atualizações assíncronas e controlam execução concorrente e serializável de regras. Este capítulo faz uma revisão dos principais trabalhos de manutenção de restrições utilizando este tipo de solução.

2.1.1 Conceitos Básicos

Várias alternativas transformam um banco de dados passivo em ativo. Segundo [CN90], as mais tradicionais são: codificar a condição que se deseja testar no próprio programa de aplicação e fazer o "polling"[1] do banco de dados, para avaliação da condição. Ambas possuem desvantagens. Veja: na primeira alternativa, o trabalho da avaliação da condição é transferido para o programador da aplicação e, na segunda, pode ocorrer desperdícios de recursos, se houver excesso de interrupções em função da freqüência estabelecida dos testes a serem realizados sobre o banco de dados. Estas desvantagens são resolvidas parcialmente com o uso de *regras* ou *gatilhos*.

[1] uma possível tradução para *polling* seria pesquisa contínua

Regras são descrições de comportamento a serem adotadas por um sistema. De um modo geral, as regras são baseadas nos seguintes componentes: *evento*, *condição* e *ação* [DBM88]. O evento é um indicador da ocorrência de uma determinada situação. Uma condição é um predicado sobre o estado do banco de dados. Uma ação é um conjunto de operações a ser executado quando um determinado evento ocorre e sua condição associada é avaliada como verdadeira. Um evento pode acionar uma ou mais regras.

Regras têm sido usadas para especificar restrições, controle de acesso e diferentes políticas de segurança e atualização. As regras também são interessantes para a manutenção de dados derivados, visões materializadas e de instantâneos, pois provêem flexibilidade para determinar quando as ações devem ser executadas (imediatamente ou quando da ocorrência de um evento específico). O uso de regras para definição de visões é destacado em [SJGP90] e [CW91]. Em [SJGP90] é utilizado um sistema de regras para suportar visões e para simular tipos de dados procedimentais. Em [CW91] são utilizadas regras de produção para a manutenção automática de dados derivados como visões.

Existem muitas derivações do conceito básico de regras definido acima, sendo as mais comuns as *regras de produção* e as *regras situação–ação*. Regras de produção já foram definidas no capítulo 1. Regras situação–ação são usadas para agrupar uma condição e sua ação associada.

Gatilhos, de acordo com [SR88], são associações de condições e ações. A execução da ação ocorre quando o banco de dados evolui para um estado que torna a condição do gatilho verdadeira. [Coh89] apresenta uma linguagem para especificar atualizações de banco de dados, consultas e gatilhos. Descreve como os gatilhos podem ser compilados em um mecanismo eficiente. [DHL90] propõe o uso de gatilhos e transações para especificar e organizar atividades de longa duração.

Grande parte das pesquisas publicadas em bancos de dados ativos discute seu suporte às restrições de integridade. Desse modo, os componentes da regra passam a possuir os seguintes significados: o evento diz respeito a um pedido de atualização; a condição, ao predicado a ser mantido; e a ação pode ser preventiva ou corretiva. Ações preventivas (o mais comum) impedem a atualização se a restrição for violada. Ações corretivas correspondem a um conjunto de operações que visa a restauração da consistência do banco de dados sem a interferência do usuário.

A maior parte das propostas de banco de dados ativo são atinentes a bancos de dados relacionais, havendo muito pouco publicado na área de bancos de dados orientados a objetos. Na maioria dos casos, o sistema

de gerenciamento de regras é implementado separado do sistema de banco de dados, ou seja, constitui uma camada adicional que não está integrada diretamente ao SGBD.

Alguns trabalhos relevantes no suporte a regras em sistemas relacionais são: [Mor83] (descreve o projeto de um SGBD ativo, focalizando algumas características básicas do banco de dados importantes para o projeto); as equações de [Mor84] (propiciam uma linguagem declarativa para expressar restrições semânticas); [WF90] (propõe a incorporação de regras de produção orientadas a conjuntos); [CW90] (propõe um sistema em que restrições especificadas numa linguagem similar à SQL podem ser transformadas em regras que detectam violações de restrições).

Em se tratando de um banco de dados relacional estendido, é destacado o trabalho de [SJGP90], que descreve o sistema de regras do POSTGRES. Este sistema de regras se propõe a suportar visões, procedimentos e restrições de integridade.

Com base no modelo orientado a objetos, existem os trabalhos de [KDM88] (descreve um mecanismo de gatilho estendido, utilizando o sistema DAMASCUS); [DBM88] (propõe regras do tipo ECA - Evento Condição Ação); [CN90] (sugere um gerenciamento de condição ativo, ou seja, um gerenciamento automático das condições a serem avaliadas sobre o banco de dados); [NQZ90] (propõe um sistema de regras para modelagem semântica implementado usando o GemStone); [UD90] (sugere a especificação e a manutenção de restrições de integridade como regras em um sistema NF_2, que, no entanto, não está integrado ao banco de dados); [MP91b] (descreve um sistema de gerenciamento de regras integrado ao SGBD orientado a objetos O_2, transformando-o em um sistema ativo de banco de dados); [MP91a] (descreve um mecanismo para garantia de integridade em um BDOO, implementado para o sistema O_2 e que usa regras de produção para manutenção de restrições), dentre outros.

A seguir, encontra-se uma descrição mais detalhada dos trabalhos citados. Para exemplificar esses trabalhos, será utilizado um esquema de dados baseado em relacionamentos onde o símbolo -- >> representa um atributo multivalorado. As relações envolvidas são: Empregado (nome, matrícula, salário, num-dep), Departamento (num, matr-gerente), Projeto (num-projeto, num-empregados) e Gerente (matrícula, num-dep).

GERENTE
SUPERVISIONA -- >> PROJETO
GERENCIA -- >> EMPREGADO

EMPREGADO
ALOCADO-PARA -- >> PROJETO

2.1.2 Morgenstern [Mor83]

Proposta

As características de banco de dados focalizadas em [Mor83] são: *descrições*, *visões dinâmicas*, *equações de restrição* e a noção de *tempo de ligação*.

As descrições são a base para o acesso aos dados e para a criação de novas entradas. Provêem os critérios definidos para visões dinâmicas que facilitam a navegação e a interação com o sistema. As equações de restrição são uma forma de representação declarativa da semântica das restrições. É destacada a importância do tempo de ligação para os métodos de acesso aos dados e para a escolha do modelo de dados.

Projeto de um SGBD Ativo

As descrições são expressas em termos do modelo de dados em uso (neste caso, o modelo ER). Utilizando o esquema de dados descrito anteriormente, uma descrição pode ser formada pela entidade EMPREGADO e pode especificar o PROJETO X para o atributo Alocado-para. Pode ser representado como: [EMPREGADO Alocado-para: X], onde [...] denota a descrição, e a primeira entrada é o tipo do objeto seguido por um par (ou mais) Nome-atributo: valor. As descrições podem ser compostas de outras descrições. Usualmente, uma descrição é construída interativamente e destina-se a *referências descritivas*, sendo a base para a noção de *visões dinâmicas*.

A referência descritiva propicia a seleção e qualificação dos objetos de interesse. Os objetos desejados são selecionados a partir da especificação parcial de informações relacionadas. Por exemplo, com o objetivo de encontrar objetos similares, pode-se fazer uma abstração ou generalização de um objeto através da inclusão de modificações na descrição. Imagine a descrição que define os empregados alocados para os projetos X e Y. Com o objetivo de encontrar todos os empregados alocados para o projeto X, basta fazer uma modificação na descrição já explicitada.

A criação de novos objetos pode ser feita baseada na existência de semelhanças a objetos já existentes. Neste caso, a descrição do objeto existente é modificada e o novo objeto é criado instanciando a descrição modificada.

Uma descrição ultrapassa as noções de consulta e recuperação, permitindo também a definição de visões dinâmicas do banco de dados. Uma visão dinâmica consiste nas ocorrências de objetos que satisfazem à descrição e possui uma extensão variável no tempo.

É possível realizar navegação através de objetos relacionados e informações seguindo relacionamentos. Por exemplo, ao se pesquisar o tipo objeto EMPREGADO, pode-se desejar visualizar o relacionamento MORA-EM, que se estende para fora do campo corrente da visão. Logo, as ocorrências relevantes do objeto RESIDÊNCIA serão trazidas para a visão.

As restrições são expressas através das equações de restrição. A manutenção direta das equações de restrição utiliza um mecanismo de gatilho. Quando ocorre uma mudança no banco de dados, o mecanismo de gatilho invoca procedimentos específicos ("demons"). Maiores detalhes sobre equações de restrição e sua manutenção podem ser encontrados em [Mor84], a seguir.

2.1.3 Morgenstern [Mor84]

Proposta

[Mor84] é continuação do trabalho proposto em [Mor83]. No trabalho de [Mor84], utiliza-se sistemas ativos para manter restrições através das equações de restrições (CEs). As CEs permitem expressar restrições semânticas que requerem consistência entre muitas relações, de forma semelhante à manutenção de relacionamentos. Cada restrição é especificada independentemente numa aplicação e é considerada uma extensão natural do esquema do banco de dados, pois aumenta a sua semântica.

As CEs permitem a representação de apenas um subconjunto das restrições que podem ser expressas através do cálculo de predicados (por exemplo, as CEs não permitem restrições que utilizem outros comparadores que não seja a igualdade). Entretanto, são consideradas pelo autor mais naturais que as fórmulas do cálculo de predicados nas quais podem ser traduzidas.

Para expressar e garantir restrições, as CEs constituem uma forma mais concisa que a escrita de procedimentos, por possuírem uma interpretação executável e por serem compiladas diretamente em rotinas para a garantia automática das restrições.

Pode-se fazer uma analogia entre equações de restrição e equações algébricas utilizando a equação $a = b + c$. Se esta equação é considerada uma restrição, sua interpretação executável pode ser visualizada como duas regras do tipo condição-ação: (1) se **b** ou **c** mudam de valor, então o valor de **a** deve ser alterado para que a igualdade seja mantida e (2) se **a** muda de valor, então **b** ou **c** ou ambos são selecionados e alterados para que a igualdade permaneça.

A restrição "Os projetos de um gerente são os mesmos projetos de seus empregados", baseada no esquema de dados já descrito, é representada usando CE da seguinte forma:

GERENTE.PROJETO == GERENTE.EMPREGADO.PROJETO

Cada lado de uma equação de restrição é uma *expressão de caminho*. Uma expressão de caminho é uma representação abreviada dos objetos de dados e dos relacionamentos do esquema.

Nessa representação os relacionamentos encontram-se implícitos. A expansão dessa restrição é representada da seguinte forma:

[(GERENTE)SUPERVISIONA(PROJETO)] ==

[(GERENTE)GERENCIA(EMPREGADO)ALOCADO-PARA(PROJETO)]

A seqüência expandida de associações da *fonte* para o *alvo* é chamada *caminho de conexão*. O componente mais à esquerda da expressão de caminho deve ser um tipo entidade que representa a fonte (no caso, Gerente) do caminho; o componente mais à direita deve ser também um tipo entidade que representa o alvo (no caso, Projeto) do caminho.

Em geral, um caminho de conexão é uma seqüência da forma:

$[(E0)R1(E1)R2...Rn(En)]$, onde Ei denota o tipo entidade e Ri denota um relacionamento entre E_{i-1} e E_i.

A tradução das equações de restrição resulta na seguinte igualdade:

$[(E0)R1(E1)R2...Rn(En)] =$

$(E0, En)|[R1(E0, E1)R2(E1, E2)...Rn(E_{n-1}, En)]$.

É definida uma álgebra para a manipulação simbólica das equações de restrição. Esta álgebra propicia a derivação de novas equações a partir das já existentes, o que torna possível uma análise simbólica das restrições e suas conseqüências.

Mecanismo de Restrições

Mudanças efetuadas no banco de dados podem afetar as equações de restrição que são automaticamente garantidas quando isso ocorre. As especificações das equações de restrição são usadas pelo compilador CE para gerar

automaticamente programas que garantam as restrições. As rotinas de garantia executam as mudanças de compensação necessárias para satisfazer as restrições afetadas por mudanças no estado do banco de dados.

A implementação do sistema de banco de dados utiliza *gatilhos* ou *demons* que são ativados quando ocorrem mudanças em determinados relacionamentos. O compilador CE acopla as rotinas de garantia de restrições geradas aos gatilhos do banco de dados para cada relação envolvida na CE. Logo, quando ocorre uma inserção, exclusão ou atualização a qualquer ocorrência das relações, a rotina de garantia é invocada automaticamente para executar a ação apropriada. Como a ativação de uma CE pode resultar em mudanças adicionais, pode ocorrer uma cadeia de ativações de muitas CEs.

Uma mudança num determinado relacionamento de um lado da CE, usualmente, pode ser compensada por mudanças do outro lado da CE para que a restrição continue sendo satisfeita. Se há mais que uma relação do outro lado da CE, escolhe-se uma delas para remover a inconsistência. Em muitos casos, as ações de compensação podem ser derivadas automaticamente das restrições. Isso é feito utilizando o símbolo "!", dado pelo usuário, que designa a relação que deve ser modificada para a garantia da restrição. Esta relação passa a ser chamada *ligação fraca* [2]. Além desta estratégia, pode-se definir explicitamente a ação a ser tomada. São as chamadas *anotações* das CEs e são representadas através de regras condição–ação.

2.1.4 Kotz, Dittrich e Mulle [KDM88]

Proposta

Em [KDM88] é apresentado um conceito generalizado de evento/gatilho como um mecanismo de suporte básico para regras semânticas. Estas são verificadas independentemente do tempo e, quando violadas, permitem a execução de ações arbitrárias. A pesquisa, neste sentido, foi motivada pela necessidade de semânticas sofisticadas em determinadas áreas de aplicações avançadas, como CAD/CAM, processamento de imagens, Inteligência Artificial e outras.

As regras semânticas englobam as regras de derivação e consistência e expressam de modo conveniente semânticas adicionais. Estas regras tiveram origem em aplicações de engenharia que, em geral, são bastante complexas. As características das regras semânticas em aplicações de banco de dados não padronizadas, como aplicações de engenharia, são:

[2] Em inglês, weak bond

- Grande número de regras complexas;
- Muitos níveis de regras locais e globais;
- Regras de verificação e/ou garantia em pontos arbitrários, freqüentemente sob o controle externo;
- Reações arbitrárias, em caso de violação de regras;
- Definição de regras de um modo algorítmico e declarativo;
- Integração de programas existentes com planos de ação;
- Definição dinâmica de regras.

Mecanismo de Regras

Existem muitos mecanismos para a verificação de integridade, porém a maioria deles não suporta os requerimentos citados. Então, propõe-se uma extensão e generalização dos conceitos de gatilhos, culminando no *mecanismo de evento/gatilho (ETM).*

O ETM é baseado em três componentes: eventos, ações e gatilhos. Todos os três podem ser definidos de modo individual e dinâmico. Um gatilho serve para associar um evento com uma ação. É representado pelo par G = (E,A). As ações podem ser executadas explicitamente ou implicitamente através de gatilhos. Os gatilhos podem ser ativados ou desativados dinamicamente.

Um evento pode ativar um número arbitrário de ações (*múltiplos gatilhos*). A seqüência de execução das ações é baseada num esquema de prioridades especificado com a definição de gatilho.

Uma ação ativada pode causar eventos que irão gerar futuras ações (*gatilhos aninhados*).

O ETM no contexto de sistemas de banco de dados provê uma interface não só para definir e excluir eventos, ações e gatilhos, mas também para causar eventos e executar ações. Esta interface pode ser explorada de dois modos: pode ser colocada disponível para usuários/programas de aplicação e pode ser usada com outros componentes do SGBD. É interessante que haja estas duas opções, pois o usuário, às vezes, tanto deseja causar eventos explicitamente quanto implicitamente, quando da ocorrência de eventos padrão. Tais eventos podem ser definidos pelo usuário usando o ETM.

Muitas vezes, deseja-se gerar eventos implicitamente quando ocorrem transições de estado do banco de dados. Para isso é utilizada uma linguagem chamada Linguagem de Definição de Evento (EDL). O EDL é um

componente adicional do SGBD que utiliza operações do ETM. Um outro componente adicional proposto é a Linguagem de Definição de Restrições (CDL), que permite a definição algorítmica de regras de consistência. Tanto o CDL quanto o EDL não fazem parte do ETM.

O ETM foi implementado como parte do DAMASCUS, um protótipo de um SGBD para projeto de VLSI. O sistema é construído como uma camada superior do sistema operacional UNIX V, usando conceitos do UNIX de comunicação de processos.

Os elementos ETM (ação, gatilho, evento) são acessados via tabelas "hash", mantidas na memória principal durante o tempo de execução. Para cada entrada na tabela de eventos, existe uma lista de gatilhos correntemente associados com este evento. A lista é ordenada por prioridade e contém ponteiros diretos para código executável das ações a serem ativadas.

2.1.5 Dayal, Blaustein e Buchmann [DBM88]

Proposta

Em [DBM88] são propostas regras Evento–Condição–Ação (ECA) como um mecanismo geral que provê capacidade ativa a um banco de dados, permitindo, assim, um melhor suporte às aplicações que requerem resposta imediata a situações críticas. Em geral, os bancos de dados convencionais não satisfazem a estas aplicações, pois são passivos.

O trabalho descrito em [DBM88] é realizado sobre um modelo de dados estendido, o HIPAC [DBB+88], e inclui construtores para representar suas regras. Estes construtores utilizam o mecanismo de regras ECA proposto, conseguindo transformar o HIPAC num SGBDOO ativo.

Mecanismo de Regras

No HIPAC as regras são tratadas como objetos. Existe uma classe de objetos do tipo regra e toda regra é uma ocorrência desta classe.

As vantagens em se tratar regras como objetos de primeira classe são: regras podem ser relacionadas com outros objetos e podem possuir atributos; e regras podem ser criadas, modificadas ou excluídas do mesmo modo que outros objetos.

Uma regra é composta de: identificador, evento, condição, ação, restrições de tempo, planos de contingência e atributos. A seguir será descrito cada componente destes.

EVENTO: O evento é uma entidade que possui um identificador e uma lista de argumentos formais tipados. Os eventos podem ser: operações sobre o banco de dados; temporais (por exemplo, o evento ocorre sempre ao meio–dia); abstratos (eventos assinalados e detectados por usuários ou outros programas); compostos (por exemplo, disjunção de dois eventos e seqüência de dois eventos).

CONDIÇÃO: A parte condição de uma regra é um objeto. Sua estrutura é descrita por duas funções: modo de acoplamento e uma coleção de consultas.

O modo de acoplamento do objeto regra indica quando uma condição deve ser avaliada relativa à ocorrência de um evento na transação ativada. Existem quatro possibilidades:

- *Imediato*, a condição é avaliada quando o evento é assinalado;
- *Diferido*, a condição é avaliada no fim da transação;
- *Acoplado mas dependente*, a condição é avaliada em uma transação separada, após o fim da transação que gerou o evento. Se a transação geradora aborta, então a condição não é avaliada;
- *Acoplado mas independente*, a condição é avaliada em uma transação separada, após o fim da transação que gerou o evento. A condição é sempre avaliada.

O segundo componente de uma condição é uma coleção de consultas. A condição é satisfeita se todas as consultas retornam respostas não vazias.

AÇÃO: A ação para uma regra é um objeto complexo. Sua estrutura é definida por duas funções: modo de acoplamento (semelhante ao definido para condição) e uma operação (por exemplo, um programa, uma mensagem para um programa externo ou um processo).

RESTRIÇÃO DE TEMPO: Corresponde a "deadlines", prioridades, urgências ou funções de valores. Não são propriedades exclusivas de regras, mas podem ser acopladas a qualquer tarefa no sistema HIPAC. São definidas como uma classe de objetos separados.

PLANOS DE CONTINGÊNCIA: São ações alternativas que podem ser invocadas sempre que a ação especificada em uma regra não pode ser executada.

ATRIBUTOS: Os atributos são propriedades adicionais das regras. Podem ser valores escalares como caracteres e inteiros ou entidades complexas compostas de outros atributos.

OPERAÇÕES SOBRE REGRAS: As operações sobre os objetos regras são: *create* (cria uma regra), *delete* (exclui uma regra), *enable* (ativa uma regra – torna-a passível de execução), *disable* (desativa uma regra) e *fire* (faz com que uma regra seja executada).

2.1.6 Widom e Finkelstein [WF90]

Proposta

Em [WF90] é proposta a incorporação de regras de produção orientadas a conjuntos em um sistema de banco de dados relacional, propiciando a definição de operações sobre o banco de dados, que são automaticamente executadas quando ocorrem determinadas condições.

Regras de produção orientadas a conjuntos são regras ativadas após a execução de um conjunto de operações e não apenas após uma operação. Além disso, as regras podem dar início à execução de um conjunto de operações sobre o banco de dados.

Widom utiliza uma linguagem de regras de produção compatível com a linguagem SQL. A ativação de uma regra resulta de operações sobre o banco de dados, geradas pela aplicação ou pelo usuário. O trabalho se detém basicamente na sintaxe da definição da regra e na semântica de sua execução.

As operações de atualização são agrupadas em *blocos de operações*. Durante a execução de um bloco de operações, conjuntos de tuplas podem ser inseridos, excluídos ou alterados. Para cada operação é definido um *conjunto afetado*, ou seja, um conjunto de tuplas "afetadas" pela operação. Por exemplo, se houver uma operação de *update*, o conjunto afetado indicará as colunas das tuplas alteradas; se houver uma operação de *insert*, o conjunto afetado conterá a tupla inserida; se houver uma operação de *delete*, o conjunto afetado conterá a tupla excluída.

O bloco de operações é executado de forma atômica. Assim, os estados intermediários não são considerados, mas apenas o estado final que resulta da execução de toda a seqüência de operações.

A execução de um bloco de operações é chamada *transição*. Uma transição

modifica um banco de dados que se encontra num estado S para um estado S'. Na verdade, o que importa é o efeito final da transição. Por exemplo, se uma tupla é alterada por muitas operações e finalmente é excluída, considera-se apenas a exclusão. Definindo formalmente, o *efeito de uma transição* é uma tripla [I, D, U], onde I é o conjunto das tuplas inseridas pela transição, D é o conjunto das tuplas excluídas pela transição e U é o conjunto de pares <tuplas, colunas> alterados pela transição.

Mecanismo de Regras

DEFINIÇÃO DA REGRA: Regras de produção são compostas por três componentes principais: um *predicado de transição*, uma *condição* opcional e uma *ação*. O predicado de transição controla a ativação de regras; a condição especifica um predicado adicional que, se verdadeiro, provoca a execução de suas ações.

Uma sintaxe do tipo SQL é usada para definição de regras de produção:

```
prod-rule-def ::= create rule NAME
                  when TRANS-PRED
                  [if CONDITION]
                  then ACTION
```

A ativação das regras de produção ocorre quando há transições de estado do banco de dados, ou seja, quando são executados blocos de operações. Os componentes da regra de produção possuem os seguintes significados. Um predicado de transição é uma lista de predicados básicos que especificam operações particulares (inserção, alteração e exclusão) sobre determinadas tabelas. A parte condição de uma regra é um predicado SQL. Este predicado tanto pode ser omitido quanto incluir operações de seleção embutidas. A omissão da parte condição da regra equivale à inclusão da sentença "if true". A parte ação de uma regra de produção é um bloco de operações (seqüência arbitrária de operações SQL). A ação também pode requisitar um *rollback* da transação corrente, isto é, um retorno ao estado inicial da transação.

Um exemplo de definição de regra utilizando o esquema de banco de dados definido no início do capítulo pode ser:
Regra: Sempre que departamentos são excluídos, todos os empregados desses departamentos devem ser excluídos.

Definição da regra:

```
when deleted from Departamento
then delete from Empregado
     where num-dep in
           (select num
               from deleted Departamento)
```

O exemplo acima ilustra a "exclusão em cascata", um método que garante integridade referencial. Nesta regra a cláusula *if* é omitida.

EXECUÇÃO DA REGRA: Regras de produção são ativadas automaticamente, em resposta a blocos de operações gerados externamente. Entre uma operação e outra do bloco de operações, regras de produção são ativadas e incluídas no fluxo de operações.

Quando há múltiplas regras a serem executadas, a interação entre regras e blocos de operações gerados externamente é: executar um bloco criando uma transição; repetidamente, executar regras ativadas (criando novas transições) até que não haja mais nenhuma ou até que ocorra um *rollback*. Este processo se repete para cada bloco de operação gerado externamente.

O exemplo abaixo ilustra a semântica de execução da regra:
Regra: Sempre que gerentes são excluídos, os seus departamentos devem ser também excluídos, assim como todos os seus subordinados.
Execução da regra:

```
when deleted from Empregado
then delete from Empregado
     where num-dep in
     (select num from Departamento
             where matr-gerente in
                   (select matricula
                       from deleted Empregado));

     delete from Departamento
            where matr-gerente in
                     (select matricula
                        from deleted Empregado)
```

O evento "deleted from Empregado" ativa a regra. A regra ativada executa a operação "delete from Empregado" que, por sua vez, gera o evento responsável pela ativação da própria regra que desencadeou tal operação.

A propriedade dessa regra de ativar a si mesma reflete sua natureza recursiva. Supondo que um bloco de operações gerado externamente exclui um ou mais empregados, então a transição correspondente identifica tuplas em Empregado, ativando a regra acima. Como não há a parte de condição (if CONDITION) na regra, a ação (then ACTION) é executada automaticamente, excluindo, assim, empregados e departamentos cujos gerentes foram excluídos pelas exclusões geradas externamente. A exclusão de novos empregados ativa novamente a regra e este processo só é interrompido quando não há mais ação de exclusão.

2.1.7 Ceri e Widom [CW90]

Proposta

Em [CW90] é estendido o trabalho anterior de regras de produção orientadas a conjuntos de [WF90]. [CW90] propõe a correção automática de estados inconsistentes do banco de dados através do uso de regras de produção. Cada restrição possui uma correspondente regra de produção. Estas regras são, então, utilizadas para detectar violação da restrição. Além disso, têm a função de iniciar operações no banco de dados que restauram a consistência.

As linguagens utilizadas para especificação das restrições e das regras são baseadas numa versão estendida do SQL. As restrições são expressas como predicados sobre o estado do banco de dados: se em um particular estado o predicado é verdadeiro, então a restrição é violada e o estado é dito inconsistente. Além disso, podem ser estabelecidas prioridades na execução das restrições.

Mecanismo de Restrições

As restrições são transformadas em regras que garantem a manutenção de restrições, emitindo ações para corrigir as violações. Em muitos casos, diversas ações podem corrigir uma dada violação de restrição e a escolha da ação mais apropriada pode depender da aplicação. Logo, para evitar problemas, as ações de compensação para cada restrição são especificadas pelo projetista da aplicação.

Uma linguagem é proposta para a especificação de restrições. A linguagem utilizada para especificação das regras de produção é a descrita em

[WF90]. Utiliza-se um sistema interativo para derivar as regras a partir das restrições. Este sistema possui a estrutura da Figura 2.1.

Abaixo será dado um exemplo da transformação de uma restrição em uma regra.

Imagine a restrição: "O salário do empregado não pode exceder duas vezes o salário médio do seu departamento. Se exceder, então remover o gerente do departamento de número 5". As tuplas que violam a restrição são expressas por uma consulta da forma:

```
R1: if exists (select * from Empregado e1
               where salario > 2*
                    (select avg(salario)
                     from Empregado e2
                     where e2.num-dep =
                           e1.num-dep)
```

A partir de uma análise sintática da restrição, detecta-se as operações que poderiam gerar violação de restrições. As operações seriam: inserção na tabela de Empregado, remoção na tabela de Empregado, atualização do salário do Empregado e atualização do número do departamento do Empregado.

A restrição R1 traduzida para a regra r1 fica da seguinte forma, segundo [WF90]:

```
r1: when inserted into Empregado
        or deleted from Empregado
        or updated Empregado.salario
        or updated Empregado.num-dep


    if exists (select * from Empregado e1
               where salario > 2*
                (select avg(salario)
                 from Empregado e2
                 where e2.num-dep =
                       e1.num-dep)
```

```
then delete from Empregado
       where matricula =
        (select matr-gerente
            from Departamento
            where num-dep = 5)
```

onde, após o **when**, encontram-se as operações que podem invalidar a restrição; após o **if**, encontra-se a condição da regra e, após o then, encontra-se a ação a ser executada em caso de violação da regra.

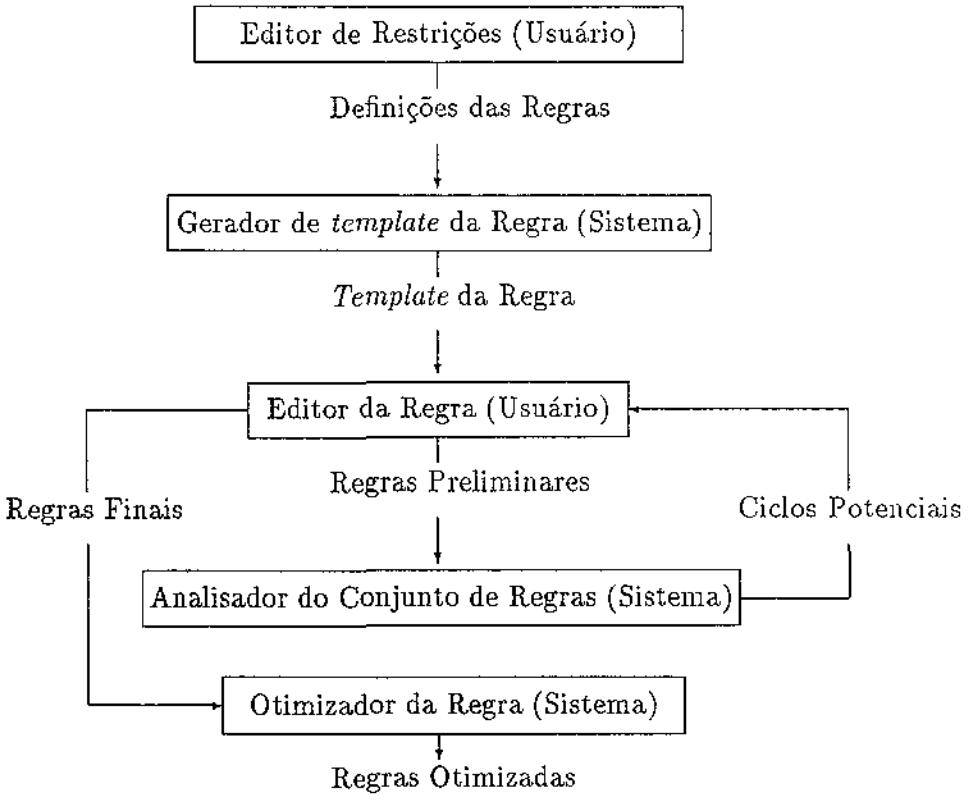


Figura 2.1: Sistema Interativo para Derivação de Regras

As porções automáticas da derivação incluem:

- Produção de esquema de regras a partir das restrições. No esquema de regras ocorre a enumeração de todas as operações que possam causar violação da restrição, além de incluir as condições das regras. O usuário é quem informa as ações das regras.

- Detecção de ciclos potenciais quando da ativação da regra. Há possibilidade de que uma regra seja ativada infinitamente, em decorrência do aumento do número e da complexidade das regras. Não obstante, o problema de garantir que um conjunto de regras termine seja indecidível, o sistema faz uma análise simples e conservadora do conjunto de regras, indicando subconjuntos de regras para os quais a ativação infinita é possível. Este componente detecta o potencial para tal comportamento, baseado nesta análise, e emite mensagens de aviso para o usuário.

- Otimização da regra. O sistema otimiza regras derivadas de restrições automaticamente. Sua semântica, entretanto, é preservada, para que a manutenção da restrição continue garantida.

Os autores garantem, baseando-se na semântica da linguagem de regra de produção e em certas suposições, no fim de cada transação as regras levam a um estado consistente.

2.1.8 Chakravarthy e Nesson [CN90]

Proposta

Em [CN90] é proposta a transformação de um banco de dados orientado a objetos passivo em ativo. Como resultado, a alternativa escolhida foi a incorporação do gerenciamento de condição ativo.

O objetivo em [CN90] é o projeto e implementação de funcionalidade ativa em um SGBDOO para posterior avaliação de desempenho. Deseja-se responder às seguintes questões básicas:

1. O gerenciamento de condição ativa é sempre melhor que o "polling" do banco de dados?

2. O "polling" do banco de dados pode ser considerado uma alternativa viável?

3. Quais as características a serem consideradas quando um SGBD passivo é transformado em ativo?

O gerenciamento de condição usando objetos ativos é comparado com o "polling" do banco de dados. O primeiro é, via-de-regra, melhor que o segundo quando a relação que está sendo monitorada é grande ou quando o tempo de resposta é importante. Entretanto, o desempenho que se esperava com o uso de objetos ativos não foi atingido. Constatou-se, também, que o "polling" do banco de dados é viável em algumas situações.

Foram propostas melhorias para a alternativa do "polling" do banco de dados. Normalmente, esta alternativa é implementada pesquisando toda a relação durante cada ciclo estabelecido. Sugeriu-se as seguintes mudanças: redução do tempo de acesso, empregando técnicas de acesso direto – indexação para as tuplas que foram modificadas – e redução do tempo gasto na avaliação da condição, armazenando informações nas tuplas que possam indicar se a condição deve ser ou não totalmente avaliada.

Projeto de um SGBD Ativo

O protótipo utiliza regras do tipo situação–ação e estas são implementadas como ocorrências da classe "active-object", a ser descrita abaixo. Parte-se do pressuposto de que um conjunto de eventos primitivos (leitura, escrita, execução de uma função, evento *clock*) podem ser detectados pelo sistema.

OBJETOS ATIVOS

Para implementação do gerenciamento de condição ativa no protótipo, são introduzidas duas novas classes de objetos – "active-object" e "activelist-object". A primeira delas engloba a funcionalidade de uma regra, incluindo informação de contexto e escopo. A segunda delas introduz suporte ao acoplamento de muitas regras a um evento específico.

IMPLEMENTAÇÃO DE MANUSEADORES DE EVENTOS

É necessário o reconhecimento de determinados eventos (leitura e escrita, no caso) para que a execução do gerenciamento de condição seja efetiva e eficiente. A alternativa considerada para implementar manuseadores de eventos foi o uso de "whoppers" que interceptam a invocação dos métodos. "Whoppers" captam código ao redor de qualquer método e constituem um mecanismo geral que possibilita o controle prioritário para execução dos métodos e a passagem de parâmetros para o próprio método atual.

FUNCIONALIDADE DO SISTEMA RESULTANTE

As novas classes introduzidas juntamente com os métodos definidos sobre elas e os manuseadores para os eventos de leitura e escrita provêem todos os mecanismos necessários para tornar o protótipo do SGBD ativo.

Os objetos ativos do protótipo suportam: gatilhos/alertas, regras múltiplas e aninhadas, encadeamento para frente e para trás, otimização na avaliação de regras e SGBD ativo.

2.1.9 Stonebraker *et al.* [SJGP90]

Proposta

Antes de apresentar a proposta de [SJGP90], é interessante observar como se deu a evolução do sistema de regras do POSTGRES [SRH90], um banco de dados relacional estendido.

Em [SRH88] foi explicada a primeira versão do sistema de regras do POSTGRES (PRS). Os implementadores do PRS tiveram os seguintes objetivos básicos: fazer com que o sistema de regras lidasse com conflitos e exceções; otimizar o processamento de regras; fazer com que o sistema de regras suportasse outros serviços do banco de dados como, por exemplo, construção de visões parciais e "read-only", controle de integridade e proteção.

O mecanismo de regras evoluiu e, em [SHP89], foram sugeridas modificações, com o propósito de melhorar a função e a usabilidade do PRS. A nova versão proposta passou a ser chamada PRS2. A funcionalidade do PRS2 englobou execução de restrições de transição e solução para o problema de atualização de visão.

Finalmente, em [SJGP90], é descrito, em detalhes, o novo sistema de regras do POSTGRES (PRS2). É mostrado que este novo sistema de regras pode ser usado para construir visões e simular tipos de dados procedimentais. É também ressaltada a grande vantagem de se armazenar em memória "cache" a parte ação das regras. Isto melhora o desempenho da execução das regras e pode ser aplicado tanto para visões materializadas como para tipos de dados procedimentais.

Resumindo, o PRS2 se propôs a simular os seguintes conceitos: visões, semânticas especiais para atualização de visões, visões materializadas, visões parciais, procedimentos, procedimentos especiais e "cacheamento" de procedimentos.

Mecanismo de Regras

A primeira versão do sistema de regras utilizava as palavras-chaves *always* e *refuse* antes de um comando POSTQUEL [3], indicando que este comando era sempre executado ou nunca executado, respectivamente. Tais palavras-chaves, junto com o comando em POSTQUEL, constituíam a sintaxe da regra. Na nova versão, utilizou-se uma representação mais tradicional para as regras, as chamadas regras de produção. A nova sintaxe passou a ser assim definida:

```
DEFINE RULE rule-name [AS EXCEPTION TO rule-name]
ON event TO object [[FROM clause] WHERE clause]
THEN DO [instead] action
```

onde **event** pode ser: *retrieve*, *replace*, *delete*, *append*, *new*, *old*; **object** é o nome de uma relação ou de um atributo de relação.

A cláusula FROM e WHERE são cláusulas normais do POSTQUEL. A parte ação da cláusula DO é uma coleção de comandos do POSTQUEL.

A semântica da regra é a seguinte: cada vez que uma tupla for consultada, alterada, inserida ou excluída será criada uma tupla CURRENT (para consultas, alterações e exclusões) e uma tupla NEW (para alterações e inserções). Se ocorre o evento estipulado, a regra é, então, verificada. Se a parte condição especificada na cláusula ON é verdadeira, então a ação é executada. Exemplo:

```
define rule exemplo1 on replace to EMP.salario
where EMP.nome = "Jorge" then do replace EMP
(salario = NEW.salario) where EMP.nome = "Carlos"
```

Quando Jorge recebe um novo salário, a regra é acionada e seu novo salário é substituído pelo salário de Carlos.

As regras no PRS2 podem ser implementadas de duas formas diferentes: em *nível de tupla* e por *reescrita de consulta*.

A implementação em nível de tupla é requisitada quando tuplas individuais são acessadas, excluídas, inseridas ou modificadas. Este sistema de regras é invocado de dois modos diferentes: quando há acesso a uma determinada tupla ou quando uma relação é aberta pelo executor. A segunda situação ocorre quando o gerenciador da regra é chamado para materializar

[3] Linguagem de Consulta do POSTGRES

tuplas de uma outra relação. A implementação em nível de tupla é interessante quando há um grande número de regras cujos eventos dizem respeito a um pequeno número de tuplas.

A outra implementação é a reescrita de consulta. Quando um usuário efetua uma consulta, a regra que se torna ativa atua indiretamente, convertendo a consulta inicial do usuário para uma forma alternativa, onde as restrições impostas pela regra ficam embutidas na consulta alterada. Esta transformação é efetuada entre o analisador da linguagem de consulta e o otimizador. A implementação por reescrita de consulta é interessante quando o evento engloba mais de uma relação e quando não há cláusulas WHERE na regra.

A escolha da implementação utilizada é baseada no número de tuplas que são referenciadas na cláusula WHERE do evento. Entretanto, se for considerado o processo de "cache" da parte ação das regras, a escolha da implementação passa a se basear em outros critérios. Este processo implica na execução antecipada da ação para as tuplas envolvidas na regra e, por fim, no "cache" dos valores resultantes.

VISÕES E PROCEDIMENTOS

As definições de procedimentos e visões no POSTGRES são praticamente semelhantes. A diferença é que o procedimento possui parâmetros e a visão não, isto é, a visão pode ser considerada um procedimento sem parâmetros.

As respectivas sintaxes da definição de procedimentos e de visões são:

```
define [updated] procedure proc-name (type-1,...,type-n) as
    POSTQUEL-command
define [updated] view view-name as POSTQUEL-command
```

Um exemplo de utilização das regras para a construção de visões é o seguinte. Considere a visão:

```
define view DEP1-EMP as
 retrieve (EMP.all) where EMP.num-dep = 1
```

A regra para construir esta visão fica assim:

```
define rule DEP1-EMP-aux on retrieve to DEP1-EMP
then do instead retrieve (EMP.OID = EMP.OID, EMP.all)
where EMP.dep = 1
```

Toda vez que o usuário faz acesso à visão DEP1-EMP, a regra acima é ativada e é feito o acesso à relação original, selecionando, assim, os campos que fazem parte da visão.

O processamento dessa regra se dá por reescrita de consulta e qualquer consulta do usuário a essa visão é alvo de modificações decorrentes da regra.

Demonstra-se, desse modo, que a definição de visões relacionais é apenas uma aplicação da implementação por reescrita de consulta sobre a regra que especifica a construção da visão. Além disso, semânticas de atualização de visões podem ser especificadas como regras de atualização, aumentando o suporte às visões.

Os procedimentos definidos no POSTGRES, do mesmo modo que as visões, são meramente aplicações adicionais do sistema de regras.

2.1.10 Nassif, Qiu e Zhu [NQZ90]

Proposta

Em [NQZ90] é proposto um modelo de dados que estende o paradigma de orientação a objetos para suportar relacionamentos e restrições. No modelo, relacionamentos e restrições são especificados de forma declarativa e constituem parte importante da semântica dos objetos.

A proposta se baseia no fato de que a semântica de atributos em sistemas OO pode ser estendida e usada como um mecanismo natural para descrever relacionamentos binários. Os relacionamentos que possuem uma semântica mais complexa são especificados via restrições. É proposta uma linguagem de restrições declarativa baseada em lógica de predicado. As vantagens desta linguagem é que as restrições ficam mais fáceis de serem expressas, mais explícitas e mais sensíveis à verificação formal. É proposta ainda a compilação das restrições para a forma de procedimento e o uso de um otimizador de tempo bastante sofisticado, com o objetivo de minimizar a sobrecarga sobre o desempenho que existe em sistemas gerais de verificação de integridade.

Mecanismo de Restrições

A proposta consiste em se ter um modelo de dados que suporte o conceito de relacionamentos. A linguagem de restrição permite expressar tanto a semântica dos relacionamentos quanto as restrições gerais. A linguagem de restrição apresentada é uma forma restrita do cálculo de predicado de

primeira ordem. Esta linguagem permite a expressão de um grande número de restrições e é traduzida automaticamente em código de procedimento.

As restrições são expressas através de relacionamentos, que são usados para modelar associações entre entidades do mundo real. Por exemplo, um relacionamento Alocado-para pode ser usado para modelar os projetos alocados para um determinado EMPREGADO. Este relacionamento pode ser modelado de muitas formas, como um atributo de EMPREGADO ou como um atributo de PROJETO. Além disso, algumas restrições semânticas podem ser acopladas aos relacionamentos. Por exemplo, o número de projetos associado a um empregado (cardinalidade) ou se um projeto pode existir quando não houver empregado alocado para ele (restrição de dependência de existência).

Os atributos no modelo proposto são objetos. Logo, possuem tanto estrutura quanto comportamento, podendo ser manipulados e consultados como qualquer outro objeto. A linguagem de restrição especifica a semântica dos atributos.

O modelo impõe que diferentes abstrações (por exemplo, generalização, agregação) sejam representadas de um modo uniforme (como relacionamentos) e que suas semânticas sejam explicitamente definidas.

2.1.11 Urban e Delcambre [UD90]

Proposta

Em [UD90] é apresentada uma ferramenta de suporte conhecida como "análise de restrições", para ser utilizada num ambiente de projeto orientado a objetos. Os autores salientam que a análise de restrições é um componente importante do gerenciamento em um sistema de banco de dados orientado a objetos. O processo de análise de restrições ajuda o projetista a entender os efeitos delas sobre manipulação de objetos, a identificar possíveis violações de restrições e a projetar alternativas para manusear violações. Uma vantagem da análise de restrições é que tanto as restrições do esquema explícitas quanto as implícitas são incluídas no processo de análise.

O artigo se detém na representação formal da análise de restrições e na identificação automática das alternativas de ações válidas para responder às violações de restrição. São apontadas cinco contribuições da análise de restrições para o projeto orientado a objetos:

1. Representação declarativa e formal das restrições explícitas e implícitas através das *cláusulas de Horn*.

2. Representação das cláusulas de Horn através dos *grafos de restrições*, com o objetivo de ordenar logicamente as cláusulas relacionadas.

3. O processo de análise de restrições ajuda o projetista a entender as suas conseqüências. Deste modo, erros podem ser detectados na fase de projeto.

4. O conhecimento sobre as restrições (por causa da análise feita) pode ser usado para formar visões do usuário de tipos abstratos.

5. A explanação das restrições permite a especificação de ações de propagação de atualização.

Como um exemplo da utilidade da análise de restrições, considere a seguinte, escrita em lógica de primeira ordem:

C1: $\forall$ e in Empregado, $\exists$ g in Gerente, $\exists$ d in Departamento, e.num-dep = d.num and d.matr-gerente = g.matrícula

Se um empregado e1 satisfaz essa restrição, um número de diferentes ações de propagação podem ser tomadas, caso o departamento seja trocado. Por exemplo, a restrição C1 pode ser satisfeita trocando o gerente do novo departamento para manter o gerente associado ao empregado. Uma ação mais plausível é simplesmente considerar que, ao se mudar o departamento, muda o gerente.

A análise de restrições ajuda na escolha da ação, mostrando todas as possíveis alternativas existentes. No caso de restrições explícitas, o usuário se encarrega de escolher uma delas.

Mecanismo de Restrições

As restrições expressas em lógica de primeira ordem são transformadas em forma normal conjuntiva. Em decorrência, fica mais fácil analisar os efeitos das operações de baixo nível do banco de dados e as alternativas que podem ser tomadas para satisfazer as restrições no caso de violação das restrições.

Exemplificando a ocorrência do processo de análise de restrições: imagine uma implicação representando uma restrição de herança que estabelece que um EMPREGADO é sempre uma PESSOA (ou seja, EMPREGADO(e) $\longrightarrow$ PESSOA(e)). Analisando esta restrição, percebe-se que quando e é inserido como EMPREGADO, já deve existir como PESSOA ou, então, uma operação deve ser invocada para inseri-lo como PESSOA. De outro modo, e não pode ser inserido como um EMPREGADO. Se e for excluído de PESSOA, o único modo de satisfazer a cláusula é exclui-lo de EMPREGADO.

Assim, nota-se que os valores verdadeiros associados com uma implicação são usados para analisar as alternativas que satisfazem a cláusula e, a partir desta análise, identificar as restrições que afetam uma determinada operação, identificar as condições que devem ser satisfeitas e identificar as alternativas de ações possíveis.

Um problema existente com o processo de análise geral das restrições é quando algumas restrições são traduzidas em um conjunto de cláusulas de Horn inter-relacionadas. Este relacionamento entre as cláusulas dificulta a análise, pois não se sabe por qual cláusula começar. Sendo assim, é proposta a organização das cláusulas em um *grafo de restrições.*

Esse grafo é formado de acordo com a relação de domínio que existe entre as cláusulas. Uma cláusula domina a outra quando seu lado esquerdo está contido ou é igual ao lado esquerdo da outra cláusula. Os grafos de restrições são usados para demonstrar propriedades que suportam uma análise ordenada das restrições.

2.1.12 Medeiros e Pfeffer [MP91b]

Proposta

[MP91b] descreve um sistema de gerenciamento de regras de produção que foi integrado ao SGBD orientado a objetos O_2, transformando-o em um sistema ativo de banco de dados. As regras encontram-se embutidas no esquema do banco de dados e, sendo assim, podem ser definidas a despeito da aplicação. O sistema proposto neste trabalho utiliza o paradigma de objetos e encontra-se operacional. Tais características o diferenciam dos demais sistemas de suporte a regras para um BDOO.

O sistema de regras do sistema O_2 possui algumas características próprias:

- está integrado ao SGBD, isto é, não é uma camada separada do sistema;
- o acesso e a ativação das regras O_2 ocorrem não só dentro dos programas como também através da interação com o usuário;
- a execução do processamento das regras se dá em dois níveis: durante o desenvolvimento do software e em tempo de execução;
- o paradigma de orientação a objetos define o mecanismo das regras, suportando herança, encapsulamento e composição de regras;

- existe independência das regras em relação às aplicações.

As regras O_2 são implementadas como objetos, sendo ocorrências de uma classe especial embutida no O_2. Deste modo, as regras são tratadas pelo SGBD da mesma forma que outros objetos. Uma regra é ativada quando ocorre um determinado evento. A regra de produção corresponde ao par <condição, ação >, onde a condição é verificada implicando na execução ou não de uma ação.

Mecanismo de Regras

REGRAS EXTERNAS EM O_2: Objetos do tipo regra são definidos como tuplas <Nome, E, C, A, T, P, S>:
Nome: nome que identifica a regra.
E(vento): eventos responsáveis pela ativação da regra.
C(onsulta): consulta O_2. Contém o predicado a ser testado com o objetivo de executar a ação.
A(ção): identifica um conjunto de operações CO_2 [4], que corresponde a uma ação a ser executada se a condição da consulta for satisfeita.
T(ipo): indica o tipo da regra, se é relacionada com o envio de mensagens (ativada por métodos) ou com o tempo (ativada pela passagem do tempo).
P(rioridade): valor que determina a ordem de execução das regras quando mais de uma é ativada pelo mesmo evento.
S(tatus): indica se a regra está habilitada ou não.

TRANSFORMAÇÃO DE UMA REGRA EXTERNA EM REGRAS INTERNAS: O sistema se encarrega de transformar cada regra externa em um conjunto de regras internas invisíveis ao usuário. As regras internas são definidas como tuplas <Nome externo, EA, C+A, P, S, T, Aut>:
EA: evento atômico ou temporal.
C+A: método CO_2 gerado pelo sistema que engloba a Consulta e a Ação.
P, S: campos de prioridade e status.
T(ipo): tipo da regra, que pode ser pré ou pós-condição, se o evento é um envio de mensagem, ou temporal.
Aut(orização): informação sobre a transação que criou a regra. É usada para determinar direitos de atualização da regra.

[4] Linguagem de programação orientada a objetos baseada em C e projetada especialmente para programar aplicações de banco de dados no sistema O_2

As regras podem ser consultadas ou atualizadas pelo usuário através de métodos que se encontram embutidos na classe especial que as define. Estes métodos podem ser invocados dentro de um programa de aplicação ou de forma interativa através da interface com o usuário.

No sistema de regras do O_2, as regras são armazenadas em listas encadeadas mantidas pelo sistema de banco de dados. Tais listas são examinadas quando ocorrem eventos específicos.

Os passos de execução de uma regra são:

1. Detecção de evento. A ocorrência de um evento é detectada e uma lista encadeada de regras é selecionada;

2. Seleção das regras internas, contidas na lista encadeada do passo 1, que não estejam inibidas;

3. Ativação dos métodos C+A das regras internas selecionadas no passo 2. Os métodos são executados de acordo com a prioridade associada.

2.1.13 Medeiros e Pfeffer [MP91a]

Proposta

[MP91a] descreve como utilizar o sistema de gerenciamento de regras de produção descrito em [MP91b] para apresentar uma solução ao problema de manutenção de integridade em sistemas orientados a objetos. Esta solução foi implementada no SGBDOO O_2 e pode ser generalizada para outros bancos de dados OO.

As restrições suportadas pelo sistema são as restrições estáticas e as dinâmicas, do tipo transição de dois estados. A manutenção da integridade é garantida executando ações de compensação determinadas pelo banco de dados e pela semântica da aplicação. As restrições são transformadas em *regras*, que são as responsáveis pelo gerenciamento da integridade. Logo, a integridade dos objetos é suportada pelos próprios objetos. Uma restrição definida para uma classe é automaticamente herdada pelas suas subclasses. E por fim, as restrições podem ser inseridas, excluídas e modificadas independentemente de qualquer aplicação.

Mecanismo de Restrições

As regras são implementadas usando o sistema de regras do O_2 e os predicados são expressos como consultas O_2 (em lógica de primeira ordem). Uma

restrição é transformada em uma regra de produção <condição, ação> da seguinte forma. Considere uma restrição estática correspondendo a um predicado P descrito em lógica de primeira ordem. Esta restrição dá origem à regra da forma $< \neg P \rightarrow A >$, onde A é uma ação que é executada quando o predicado (restrição) P não é satisfeito. No caso da restrição dinâmica, esta é transformada em um conjunto de restrições estáticas baseado no estado inicial e final da transação. Sendo assim, a regra de produção seria uma tupla, como definido na seção anterior, onde os atributos C(onsulta) e A(ção) corresponderiam ao predicado (P) e a ação (A). A execução do par <C, A> pode ser ativada por diferentes eventos de atualização. Assim, o passo seguinte é determinar todos os eventos que requerem verificação da restrição. Isso é feito em duas etapas. Na primeira, o predicado da consulta é analisado e todos os objetos e nomes de classes referenciados são considerados fontes de violação de restrições; na segunda etapa, o esquema do banco de dados é examinado para identificar todos os métodos que enviam mensagens para as fontes detectadas na etapa anterior. Este processo é manual.

Após essas etapas, os objetos do tipo regra resultantes são inseridos no banco de dados. Se necessário, o mecanismo de regras cria regras adicionais para garantir a herança de restrições. A este fenômeno dá-se o nome de *propagação de restrições por herança*. Então, uma restrição representada como uma regra de produção pode levar a um conjunto de regras O_2, pois a um evento pode corresponder várias regras diferentes.

Pode haver muitas regras ativadas por um mesmo evento. A solução adotada é a indicada por [CW90]: "executá-las conforme a prioridade associada a cada regra".

2.1.14 Quadro Comparativo

A tabela a seguir mostra um quadro comparativo dos artigos detalhados neste capítulo, enumerados na primeira coluna. A segunda coluna indica o modelo de dados correspondente; a terceira coluna, os componentes básicos do sistema de restrições; a quarta coluna, a linguagem utilizada para definição de restrição (e possivelmente de regras); a última coluna, se o mecanismo proposto encontra-se automatizado ou não.

Artigo	Modelo de Dados	Componentes	Linguagem Restrição/Regra	Automatização
[Mor83] [Mor84]	E/R	Equações de Restrição (CE)	Sim, CEs	Sim
[KDM88]	OO (DAMASCUS)	Eventos/gatilhos (E, G, A)	Sim, CDL (Ling. Def. Restrição)	Sim
[DBM88]	OO (HIPAC)	Regras(E, C, A)	Não	Sim
[WF90] [CW90]	Relacional (STARBURST)	Regras(E, C, A)	Sim, similar à SQL	Sim
[CN90]	OO	Regras situação-ação	Não	Sim
[SJGP90]	Relacional estendido (POSTGRES)	Regras(E, C, A)	Sim, modificações na linguagem POSTQUEL	Sim
[NQZ90]	OO (GemStone)	Relacionamentos	Sim, baseada lógica de predicado	Sim
[UD90]	NF_2	-	Sim, lógica de 1ª ordem	Não
[MP91b] [MP91a]	OO (O_2)	Regras(E, C, A)	Sim, lógica de 1ª ordem	Não

Capítulo 3

Taxonomia de Restrições no Modelo OO

3.1 Introdução

Este capítulo apresenta uma taxonomia proposta nesta tese para classificar restrições estáticas de integridade, que servirá de base aos capítulos seguintes.

Para melhor exemplificar a taxonomia, será utilizado um banco de dados de uma empresa mostrado na Figura 3.1. A notação adotada é baseada no SGBD O_2. Maiores detalhes sobre a notação podem ser encontrados em [Alt89].

No esquema apresentado:

- O relacionamento entre uma *superclasse* e uma *subclasse* é introduzido com o termo **inherits**.
- Cada componente de uma classe é denotado por nome: tipo, onde nome é o nome do componente e tipo é um tipo atômico ou composto.
- A classe Pessoa possui como subclasses Cliente e Empregado. Logo, ambas herdam seus atributos e métodos.
- A classe Cliente-Empregado estabelece uma herança múltipla, posto que tem como superclasses Cliente e Empregado. Sendo assim, herda os métodos e atributos tanto de Cliente quanto de Empregado.

Informações estruturais/herança

Class *Pessoa* **type**
tuple *(nome: String, datanasc: Data, endereço: Endereço)*

Class *Empregado* **inherits** *Pessoa* **type**
tuple *(cargo: Integer, tempo-serviço: Integer, dependente: set(Pessoa),*
salário: Dinheiro, dep: Departamento)

Class *Cliente* **inherits** *Pessoa* **type**
tuple *(crédito: Dinheiro, status: String)*

Class *Departamento* **type**
tuple *(nome: String, pessoal: set(Empregado), gerente: Empregado)*

Class *Cliente-Empregado* **inherits** Cliente, Empregado

Informações comportamentais

method *atualizar-datanasc (datanasc-atual: Data)* **in class** Pessoa
method *demitir (emp: Empregado)* **in class** *Empregado*
method *contratar (emp: Empregado)* **in class** *Empregado*
method *atualizar-salário (salatual: Dinheiro)* **in class** *Empregado*
method *atualizar-dep (depant, depatual: Departamento)*
in class *Empregado*
method *atualizar-gerente (geratual: Empregado)*
in class *Departamento*
method *atualizar-salário (salatual: Dinheiro)*
in class *Cliente-Empregado*

Figura 3.1: Esquema do Banco de Dados de uma Empresa

- Um método definido numa superclasse pode ser redefinido na subclasse. A isso dá-se o nome de **sobreposição** (por exemplo, o método atualizar-salário da classe Cliente-Empregado).

OBS.: As classes Dinheiro e Data não foram definidas no esquema, por terem sido tratadas do mesmo modo que os tipos de dados atômicos. Além disso, não houve preocupação em se definir o corpo dos métodos do esquema porque isso não é relevante para suportar as definições posteriores.

3.2 Taxonomia

A taxonomia das restrições estáticas para o modelo de dados orientado a objetos considera: dependência interclasse, dependência intraclasse, restrição interobjeto, restrição intra-objeto, restrição sobre métodos, restrição global e restrição local. Abaixo serão descritas tais restrições, utilizando como exemplos as classes do esquema mostrado na Figura 3.1. A figura 3.2 mostra a taxonomia adotada.

DEPENDÊNCIA INTERCLASSE

São restrições entre classes diferentes. As principais causas da dependência interclasse são herança e composição. No caso de herança, a ligação existente entre as classes Empregado e Pessoa demonstra esta dependência, uma vez que uma inclusão em Empregado implica na necessidade de um objeto em Pessoa. Na composição, a dependência interclasse corresponde ao fato de que um objeto composto agrega vários componentes. O exemplo a seguir tem analogia com a dependência de inclusão (ID) do modelo relacional existente entre o atributo gerente da classe Departamento e a classe Empregado. Sendo gerente pertencente à classe Empregado, uma inclusão no atributo gerente depende da ocorrência correspondente na classe Empregado.

Um outro exemplo a ser considerado e que não diz respeito à herança e composição é a restrição "o salário de um empregado cuja idade é superior a 50 anos é no mínimo 300.000". Existe uma dependência interclasse entre as classes Data e Dinheiro, estabelecida dentro da classe Empregado, dado que os atributos datanasc e salário da classe Empregado atingidos pela restrição pertencem, respectivamente, a estas classes.

DEPENDÊNCIA INTRACLASSE

São restrições dentro de uma mesma classe. O exemplo de dependência interclasse utilizado acima também pode ser considerado um exemplo de dependência intraclasse, já que existe uma dependência dentro da própria classe Empregado entre os atributos datanasc e salário. Entretanto, existem muitas situações de dependência intraclasse que não são interclasse. Imagine a restrição "se o tempo de serviço de um empregado é superior a 10 anos, então este empregado não pode ser do cargo estagiário". Neste caso, existe uma restrição entre os atributos cargo e tempo-serviço da classe Empregado. Como os dois atributos, neste exemplo, pertencem a tipos atômicos e não a classes definidas pelo usuário, não existe dependência interclasse entre eles.

RESTRIÇÃO INTEROBJETO

A restrição interobjeto diz respeito às restrições entre objetos de uma mesma classe ou no caso de duas classes distintas, entre objetos específicos. Um exemplo é a restrição: "Os salários dos gerentes Carlos e Pedro devem ser iguais". Os objetos pertencem a mesma classe (Empregado). Entretanto, a restrição "Carlos está vinculado ao departamento D1" é uma restrição entre os objetos, Carlos e departamento D1, pertencentes a classes distintas, porém inter-relacionadas pela classe Empregado. Um exemplo de restrição interobjeto entre duas classes não relacionadas (Empregado e Carro) é "O salário de Carlos não pode ultrapassar o preço total de um carro VW".

RESTRIÇÃO INTRA-OBJETO

A restrição intra-objeto diz respeito às restrições entre componentes do tipo de uma classe. Sendo assim, pode-se considerá-la restrição intraclasse. Um exemplo é a restrição: "O tempo de serviço determina o salário do empregado Pedro", que estabelece uma restrição entre os componentes do tipo da classe Empregado: tempo-serviço e salário.

RESTRIÇÃO SOBRE MÉTODOS

A restrição sobre um método determina a condição para sua execução. Esta restrição pode ser estabelecida como pré ou pós-condição. Estas condições podem ser relativas à premissa de acesso/modificação dos dados, relativas à

ordem dos métodos e relativas ao tempo.

Em relação ao acesso, faz sentido apenas a restrição estabelecida como precondição. Um exemplo é a restrição: "o método atualizar-salário só pode ser enviado a um empregado se o tempo de serviço dele for maior que seis meses".

Em relação à ordem dos métodos, a restrição pode ser estabelecida em termos de pré e pós-condição. Uma precondição para um método X é "o método X só será ativado se o método Y tiver sido ativado anteriormente." Uma pós-condição para um método X é "o método X no fim da sua execução deve ativar o método Y".

Em relação ao tempo, é interessante que se estabeleça a restrição como precondição. Este tempo pode ser absoluto ou relativo. Por exemplo, "o método X será ativado todo dia às 12 h (tempo absoluto)" ou "o método X será ativado de três em três horas (intervalo de tempo)".

RESTRIÇÃO GLOBAL

É uma restrição estabelecida para todo o banco de dados, ou seja, para todas as aplicações ativas. Por exemplo, toda campo data do banco de dados deve ser válido.

RESTRIÇÃO LOCAL

É uma restrição definida localmente para uma aplicação específica. Sua verificação é executada sempre que a aplicação está ativa. Por exemplo, o salário de um empregado não pode ser zero.

3.3 Restrições no Modelo Relacional

Para mostrar como a taxonomia proposta é aplicável a sistemas relacionais, esta seção faz um mapeamento das restrições de integridade do modelo de dados relacional para o modelo orientado a objetos.

Uma taxonomia comum no modelo relacional é a que divide as restrições em inter-relação (por exemplo, chave estrangeira) ou intra-relação (por exemplo, FD). Pode-se afirmar, por analogia, que existem dependências deste tipo no modelo OO proposto. De fato, a inter-relação corresponde à dependência interclasse e, a seguinte, à dependência intraclasse.

Todavia, esse mapeamento das restrições do modelo relacional para o

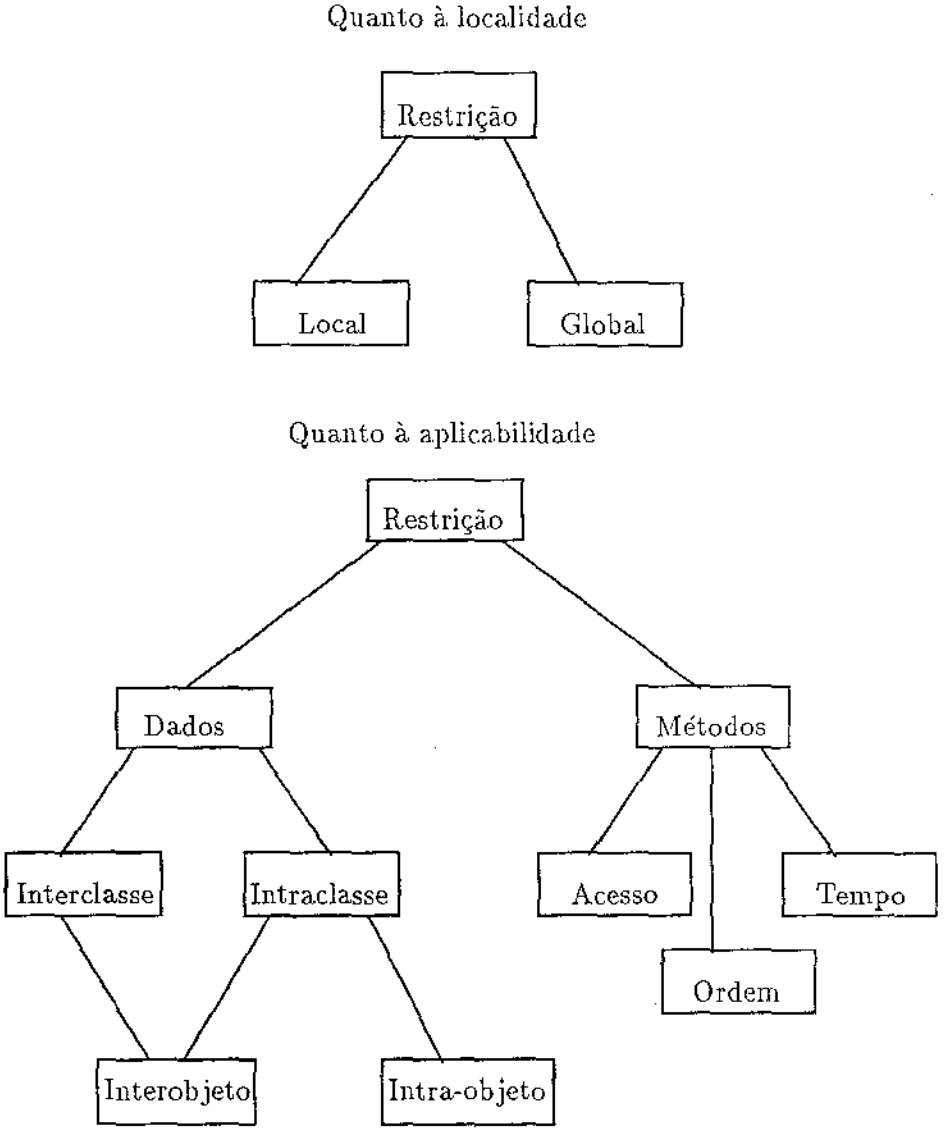


Figura 3.2: Representação da Taxonomia de Restrições

modelo OO não cobrirá todas as restrições existentes no modelo OO. Isso ocorre em virtude da riqueza deste último, que engloba não apenas os dados, mas também os métodos sobre esses dados. Assim, como já visto pela taxonomia proposta, além das restrições sobre os dados, devem ser consideradas as restrições sobre os métodos. Estas últimas restrições não podem ser portáveis de modelos anteriores, visto que o conceito de comportamento só existe no modelo OO. Abaixo serão analisadas restrições clássicas do modelo relacional segundo o modelo OO, exemplificando com as classes do esquema de dados, já definido.

DEPENDÊNCIA DE INCLUSÃO

Pode-se considerar que no modelo OO existe dependência de inclusão entre subclasse/superclasse estabelecida pela herança, entre classe/classe estabelecida pela composição e entre componentes do tipo de uma classe.

Imagine uma ID no modelo relacional entre as relações Empregado e Dependente. A exclusão de um empregado implica na exclusão de todos os seus dependentes. A inclusão de um dependente implica na existência de um empregado que corresponda àquele dependente.

No modelo OO, esse tipo de restrição é mantida automaticamente, por causa da identidade de objetos, já definida. Considere a classe Empregado do esquema de dados da Figura 3.1 e os seguintes objetos desta classe:

- <Antônio Barreto, 010360,..,..,..,<João Barreto, 050590,..>,
  <Maria Barreto, 070688,..>, 500000,
  <Recursos Humanos,...,...>>
- <Sílvia Barreto, 291165,..,..,..,<João Barreto, 050590,..>,
  300000, <Proc. de Dados,...,...>>

Se considerada como no modelo relacional, a atualização do objeto <João Barreto, ...>, pertencente à classe Pessoa, implicaria na propagação desta atualização para os objetos <Antônio Barreto, ...> e <Sílvia Barreto, ...>, pois ambos o referenciam. Contudo, no modelo OO, na verdade, o que existe é um ponteiro para o objeto e não uma cópia explícita do objeto. Sendo assim, a atualização do objeto <João Barreto, ...> não precisa ser propagada, pois a sua identificação é feita de forma física pelo "oid" e não de forma lógica.

DEPENDÊNCIA FUNCIONAL

É razoável sugerir que existam FDs no modelo OO usando a mesma definição do modelo relacional, como por exemplo restrição de chave. Às vezes deseja-se estabelecer que a matrícula do empregado seja única entre os objetos daquela classe.

DEPENDÊNCIA MULTIVALORADA

Esse tipo de restrição pode ser eliminada usando noções de aninhamento de relações durante a modelagem dos dados. Sendo assim, a dependência multivalorada deixa de existir explicitamente no modelo OO.

3.4 Considerações

O fato do modelo OO considerar composição e herança, o distingue de outros modelos. Estas duas propriedades geram implicações quanto às restrições de integridade.

A composição é pertinente à formação de um objeto, tendo como componentes outros objetos. Logo, uma restrição sobre um atributo de uma classe A pertencente a outra classe B deve ser ativada não só quando a classe B é atualizada diretamente, mas também quando há modificações na classe A que influenciam a classe B. A possibilidade de se definir progressivamente objetos complexos acarreta, desta forma, o problema de verificar o fechamento do conjunto de restrições aplicado a uma dada classe ou conjunto de classes. Nos modelos anteriores, este problema praticamente não existia, já que não permitiam flexibilidade em reestruturação incremental. Problemas associados são a consistência de um conjunto de restrições e de interferência entre restrições.

O problema de interferência entre restrições é muito complexo e já foi analisado para alguns casos particulares no modelo relacional. (Esta dissertação não irá analisar este problema, considerando que cabe ao projetista determinar um conjunto coerente de restrições).

A herança traz conceitos de subclasse e superclasse, onde a subclasse, classe mais especializada, herda todas as propriedades da classe mais geral, a superclasse. Estendendo este conceito, pode-se falar de *herança de restrições*, ou seja, uma restrição de uma classe é herdada por todas as suas subclasses. Entretanto, podem ocorrer situações em que as subclasses não

queiram herdar determinadas restrições. Sendo assim, um outro ponto a considerar é a *negação de restrições herdadas*, ou seja, uma forma de evitar que a restrição se mantenha na subclasse. Um exemplo de negação de restrição seria o seguinte: imagine uma classe Empregado e a subclasse Diretor. A restrição: "o salário de um empregado não pode ser superior a 400.000" não se aplica à classe Diretor. Esta herda os dados e métodos da classe Empregado, mas não deseja herdar a restrição definida acima. Esta hipótese corresponde à existência de exceções em que subclasses não herdam todas as propriedades dos seus ancestrais. Um exemplo desta filosofia, para modelagem de objetos, é proposta em [Bor85]. (Para efeito desta tese, a idéia de exceções não será considerada.)

No caso de herança múltipla, podem surgir alguns problemas. Por exemplo, a classe Cliente-Empregado que possui como superclasses Cliente e Empregado herda, a princípio, as restrições definidas pelas suas superclasses. Se houver conflitos entre estas restrições herdadas, pode-se optar por redefinir a restrição na classe Cliente-Empregado ou então por herdar apenas uma das restrições conflitantes.

Capítulo 4

Transformação de Restrições em Regras

4.1 Introdução

A proposta desta dissertação é que restrições sejam mantidas em bancos de dados OO utilizando regras. Para isto, são necessários ao menos dois componentes em um SGBDOO:

- mecanismo que, dada uma restrição, determina as regras correspondentes;
- mecanismo de controle das regras.

Na bibliografia correlata apenas [CW90] e [Mor84] apresentam algoritmos de transformação de restrições em regras. [MP91a] indica como fazê-lo, usando a idéia de [CW90]. Estas opções existentes para o tratamento automático de restrições são representativas do que é sugerido na literatura. As abordagens de [CW90, Mor84], sobre o tratamento de restrições de integridade, por este motivo, foram analisadas mais detalhadamente e, no final, foi feita uma opção por uma delas, para servir de base ao trabalho proposto. Esta introdução mostra tal análise e justifica a escolha da metodologia básica para a transformação de regras. O restante do capítulo propõe um algoritmo que permite esta transformação.

As equações de restrição propostas em [Mor84] provêem uma linguagem declarativa para especificar restrições. O conjunto de restrições que pode ser expresso através das equações de restrição é apenas um subconjunto das

restrições que pode ser expresso usando predicados arbitrários (por exemplo, nas equações de restrição, o único operador permitido é o de igualdade). As equações de restrição propiciam uma forma bastante simples de expressão das restrições. Entretanto, utilizar esta abordagem, gera algumas dificuldades. A primeira delas é a limitação do conjunto de restrições que podem ser representadas. Algumas extensões seriam necessárias para permitir outros operadores e continuar a manutenção automática de restrições. A segunda dificuldade é o fato de ser utilizado um modelo com relacionamentos sobre o modelo relacional enquanto que o trabalho aqui proposto é sobre o modelo orientado a objetos. Existe, assim, uma grande dificuldade em utilizar as equações de restrição por causa da inexistência de relacionamentos explícitos no modelo orientado a objetos. Para ilustrar este problema, imagine o relacionamento Alocado-para entre os objetos EMPREGADO e PROJETO, que identifica os projetos para os quais os empregados estão alocados. No modelo relacional, este relacionamento é representado como uma tabela separada fazendo a ligação entre EMPREGADO e PROJETO. No modelo orientado a objetos, relacionamentos podem ser simulados através de composição. No caso, o relacionamento pode ser representado como um atributo de EMPREGADO e/ou de PROJETO, mas não há possibilidade de definir cardinalidade e não pode ser facilmente expresso como uma equação de restrição.

Uma forma para resolver o problema de relacionamentos no modelo orientado a objetos seria utilizar a proposta de [NQZ90]. Nassif, em [NQZ90], propõe uma extensão do modelo orientado a objetos para suportar relacionamentos e restrições. Os atributos, as entidades e os relacionamentos são tratados como objetos, e os atributos são usados para definir relacionamentos. A modelagem dos dados é feita baseada no modelo ER. A desvantagem da proposta de Nassif é a não utilização do paradigma de orientação a objetos na sua essência, o que torna o processo não muito natural, visto que utiliza a modelagem ER e modifica a semântica do modelo OO para introduzir relacionamentos.

O trabalho de [CW90] propõe um sistema interativo incorporado a um banco de dados relacional para derivar regras a partir de restrições. As restrições são expressas em uma linguagem baseada em SQL. A partir das restrições, deseja-se derivar automaticamente o conjunto de operações que podem causar violação de restrições. Isso é feito através de uma análise puramente estática. Como não é feita uma análise semântica da restrição, o conjunto de operações detectado pode vir a incluir operações que não possuam o potencial para violar a restrição. Contudo, isso pode afetar apenas

a eficiência do sistema: os autores garantem que, com seu método, interceptam corretamente todas as condições de violação. Embora a proposta seja voltada para o modelo relacional, a análise sintática da restrição, que é feita para detecção dos eventos responsáveis pela ativação da regra (ou seja, das operações que podem invalidar a restrição), mostra-se interessante para o modelo orientado a objetos.

Este capítulo estende a proposta de [MP91a], de transformação de restrições em regras. Em [MP91a], esta transformação é manual, sendo baseada em idéias de [CW90]. Uma regra de produção é composta basicamente de evento, condição e ação. A condição pode ser derivada diretamente da restrição e a ação é determinada pelo usuário. Entretanto, a detecção dos eventos é o passo mais complexo. No caso de sistemas orientados a objetos, estes eventos estão associados a *envio de métodos*. Isto complica enormemente a definição de todos os eventos que estejam associados à violação de uma possível restrição, por dois motivos principais:

- Sobrecarga e polimorfismo associados ao envio de métodos que impedem que se determine de forma estática a cadeia de execução e mesmo a classe do receptor;
- Riqueza de funções e nomes associados à dinâmica de uma aplicação.

Estes dois problemas não ocorrem na maioria dos sistemas ativos de controle de restrições existentes na literatura, já que se trata normalmente de sistemas relacionais. Assim, algoritmos, como os propostos por [CW90, Mor84], não se aplicam no caso de sistemas orientados a objetos.

[MP91a] propõe que a detecção dos eventos seja feita em duas etapas. A primeira delas corresponderia à *análise de caminho* (análise sintática da restrição para a derivação dos eventos), semelhante ao descrito em [CW90]. O predicado de consulta, ou seja, a condição , é analisado e todos os objetos e classes mencionados são considerados fontes de violação de restrições. A determinação destas fontes deve ser automática, pois necessita apenas de uma análise sintática da sentença. Todavia, existe uma diferença em relação ao trabalho de Widom, onde só a análise puramente sintática é suficiente, em virtude de utilizar o modelo relacional. Como o modelo utilizado por [MP91a] é o modelo orientado a objetos, é necessária uma análise mais aprofundada da restrição por causa da existência dos métodos. A segunda etapa é, portanto, um exame do esquema do banco de dados para identificar todos os métodos que enviam mensagens para as fontes detectadas na etapa anterior.

A detecção dos eventos tem por objetivo evitar a verificação desnecessária de restrições que não são violadas pela atualização em questão. Como o problema de encontrar uma solução exata para o problema é indecidível, aqui se apresenta um algoritmo que proporciona um superconjunto da solução.

As seções que seguem propõem uma solução automática para a proposta de [MP91a], que inclui refinamentos à proposta original.

O capítulo define a seguir uma linguagem de restrição que utiliza lógica de primeira ordem sem negação. Esta linguagem pode ser usada para definir restrições estáticas inter e intraclasse, inter e intra-objeto e sobre métodos, em sistemas orientados a objetos, segundo a taxonomia proposta.

As modificações principais introduzidas com relação à proposta de [MP91a] são as seguintes:

- A determinação de eventos que podem violar uma regra é automatizada (enquanto em [MP91a] era manual, sem especificação de detalhes);
- Enquanto as restrições abordadas por [MP91a] são relativas a dados, este trabalho considera também restrições sobre métodos;
- A propagação de eventos e restrições para subclasses é feita pelo algoritmo (o que não ocorria em [MP91a], onde tal propagação era deixada ao sistema de regras);
- Este trabalho especifica as informações que se deve ter a respeito do comportamento de objetos para realizar esta automatização (não definida em trabalhos anteriores);
- Finalmente, e também devido ao item anterior, a proposta implementada não depende de um modelo específico orientado a objetos e pode ser incorporada a qualquer sistema que use um modelo baseado em classes com tipo, enquanto [MP91a] usava um mecanismo inserido no SGBD O_2. Neste último, a determinação de herança de restrições era deixada ao sistema, o que não ocorre em outros SGBD. Deste modo, o algoritmo proposto é de uso geral.

4.2 Linguagem de Restrição

Esta seção propõe uma linguagem de especificação de restrições estáticas para sistemas orientados a objetos, baseada em lógica de primeira ordem,

que permite descrever restrições inter e intraclasse, e inter e intra-objeto. Restrições sobre métodos só podem ser expressas quando se referem a pré ou pós-condição de execução de métodos.

As restrições de integridade são construídas usando objetos e classes do esquema e operadores gerais. Os operadores são os seguintes:

- operadores aritméticos: +, −, *, /
- comparadores de atributos: =, <, >, <=, >=, <>
- comparadores de objetos: ==, !=
- operadores booleanos: **e**, **ou**
- operador lógico: →
- quantificadores: **existe**, **paratodo**
- operadores de agregação: **sum**, **min**, **max**, **card**
- operadores de pertinência: **contem**, **estacontido**
- operadores de execução de métodos: **pre**, **pos**

Note que há duas possibilidades de comparação: comparação de valores (igual à comparação-padrão) e comparação de objetos. Ambas são consideradas separadamente em vários sistemas orientados a objetos, já que o conteúdo (valor) de dois objetos pode ser o mesmo, mas os objetos podem ser diferentes ("oid" diferente). Assim "==" se refere à igualdade de "oids" e "!=" à desigualdade de "oids". Para efeito de detecção de eventos, não há diferença no tratamento destes operadores , como será visto. Os operadores se diferenciam na verificação da restrição (ou seja, o campo Condição) e, portanto, a consulta a ser realizada pela regra precisa usar operadores diferentes, segundo as diferentes linguagens de consulta existentes.

Outra observação se refere aos operadores **pre** e **pos**, onde os operandos denotam envio de métodos. Os operandos são, então, os próprios eventos a serem detectados pelo algoritmo.

Exemplos de restrições:

- R1: (**paratodo** p **em** Pessoa; p.idade <= 130 **e** p.idade >= 0).
- R2: (**paratodo** m **em** Motorista; m.idade >= 18).

- R3: (**paratodo** p **em** Pessoa; **card**(p.crianças) < 20).
- R4: (**paratodo** v1, v2 **em** Veículos; v1.número <> v2.número **ou** v1 = v2).
- R5: (**existe** e1 **em** Empregado; e1.dep.nome = 'Pessoal').
- R6: (**paratodo** v **em** Veículos, **existe** m **em** Motorista; m.carro = v).
- R7: **pre**(atualizar-salário → e1 **em** Empregado) (**paratodo** e1 **em** Empregado; e1.salário <> 0).

Abaixo é dada uma gramática para a linguagem de restrições. A linguagem proposta é uma variação da lógica de primeira ordem. A gramática é LL(1), isto é, permite analisar uma cadeia da esquerda para a direita, produzindo uma derivação esquerda e verificando apenas um símbolo da cadeia de entrada para decidir qual produção a ser aplicada.

A sintaxe da linguagem de restrições será dada em FNB estendida. Os não-terminais da gramática são nomes colocados entre parênteses angulares < e >, como, por exemplo, <predicado>.

GRAMÁTICA

0. <restrição> ::= (<predicado>) /
   <ordem-método>(<predicado>)
1. <ordem-método> ::= <op-método>(<método> {,<método>} →
   <var> {,<var>} **em** <domínio>)
2. <predicado> ::= <quantificador>; <seleção>
3. <quantificador> ::=
   existe <var> {,<var>} **em** <domínio> [, <quantificador>]/
   paratodo <var> {,<var>} **em** <domínio>
   [, <quantificador>]
4. <seleção> ::= <seleção-1> <implica> <seleção-1>/
   <seleção-1>
5. <seleção-1> ::= <condição>/
   <condição> <conector> <seleção-1>/
   (<seleção-1>)
6. <condição> ::= <expressão>
7. <expressão> ::= <exp-simples> <comparador> <exp-simples> /

<exp-simples>

8. <exp-simples> ::= [+|−]<termo> {(+|−) <termo>}
9. <termo> ::= <fator> {(∗|/)<fator>}
10. <fator> ::= <var-exp> /
    <string> /
    <numero> /
    (<expressao>) /
    <exp-agregação> /
    <exp-pertinência>
11. <var-exp> ::= <var>.<atributo>
12. <exp-agregação>::= <op-agreg> (<var-exp>)
13. <exp-pertinência>::= <op-pert> (<var-exp>, <var-exp>)
14. <conector> ::= e / **ou**
15. <implica> ::= →
16. <comparador> ::= <comp-atributo> / <comp-objeto>
17. <comp-atributo> ::= = / <> / > / < / >= / <=
18. <comp-objeto> ::= == / !=
19. <op-método> ::= **pre** / **pos**
20. <op-agreg> ::= **sum** / **min** / **max** / **card**
21. <op-pert> ::= **contem** / **estacontido**
22. <var> ::= <identificador>
23. <domínio> ::= <identificador> *
24. <atributo> ::= <identificador> **
25. <método> ::= <identificador> ***
26. <identificador> ::= <letra> {<letra> | <dígito>}
27. <número> ::= <dígito> {<dígito>}
28. <letra> ::= a/ b/ c/..../z
29. <dígito> ::= 0/1/2/...../9
30. <string> ::= '<componentestring> {<componentestring>}'
31. <componentestring> ::= " "/qualquercaracterexcetoapóstrofe

* nome de classe
** nome de atributo de classe ou de método
*** nome de método

4.3 Proposta do algoritmo

Nesta seção será descrito o algoritmo proposto para determinar os eventos (atualizações) que podem causar violação de restrição, sendo usada a terminologia a seguir.

4.3.1 Terminologia Utilizada

- *Esquema*, a definição encontra-se no capítulo 1.
- *Atributos do método* são os campos que podem ser atualizados por um método, não necessariamente especificados na assinatura.
- *Atributos da classe* são os campos componentes dos objetos que pertencem àquela classe.
- *Atributos fontes de violação de restrições* são os atributos da classe cuja atualização pode violar uma dada restrição.
- *Característica extraída de um atributo* são os componentes parciais de um atributo. Por exemplo, o atributo x.a.b.c possui como característica x.a e x.a.b, onde a, b e c não são nomes de métodos.

4.3.2 Idéias Centrais do Algoritmo

O algoritmo descrito tem por objetivo a detecção dos eventos que podem levar à violação de uma restrição. Estes eventos correspondem ao par <classe, método>, onde o método sempre pertence à classe especificada no par. Por exemplo, um dos eventos que podem acionar uma restrição sobre o salário dos empregados é o evento <Empregado, atualizar-salário>, isto é, cada vez que o método atualizar-salário é invocado por um objeto da classe Empregado, a restrição sobre o salário dos empregados deve ser verificada.

Para conseguir detectar automaticamente os eventos a partir de uma restrição, os seguintes passos são executados:

1. A restrição escrita na linguagem proposta é analisada e as classes referenciadas na restrição são determinadas (por exemplo, na restrição acima a classe identificada é Empregado).
2. São identificados os métodos pertencentes a essas classes que podem violar a restrição, quando invocados. Para isso, é verificado se os

métodos das classes determinadas podem modificar alguns dos atributos que se encontram na definição da restrição. No caso da restrição acima, o algoritmo verifica se o método atualizar-salário se referencia ao atributo salário. Se isso acontecer, a classe juntamente com o método detectado constituem um evento. Quando este evento ocorre, a verificação da restrição correspondente se torna obrigatória.

3. Os pares <classe, método> obtidos no passo anterior constituem os eventos-bases. O esquema do banco de dados é então percorrido para determinar que outros eventos podem violar a restrição, tendo em vista o grafo de herança. Em outras palavras, o sistema implementa um mecanismo de herança de restrições.

Note-se que uma dada restrição pode ser violada por mais de um evento.

4.3.3 Algoritmo

As informações de entrada para o algoritmo englobam a *restrição* e as *informações do esquema do banco de dados* (*grafo de composição e herança*, métodos e atributos por eles modificados). Para facilidade de compreensão do algoritmo, as informações sobre o esquema se encontram em tabelas. À saída, o sistema indica um *conjunto* de regras <E, C, A> que deve ser verificado para aquela restrição, que é expressa na linguagem de restrição proposta.

As informações do esquema do banco de dados necessárias são as contidas nas tabelas T1 e T2, a seguir:

TABELA T1

Classe	Atributo: tipo
.	.
.	.

TABELA T2

Classe	Método	Atributo: tipo
.	.	.
.	.	.

- Tabela T1: classes do esquema e seus respectivos atributos e tipos/classes[1] a que pertencem (grafo de composição).

  Essa informação é necessária, pois, a partir dela, pode-se saber quais as classes cuja atualização pode afetar uma determinada restrição.

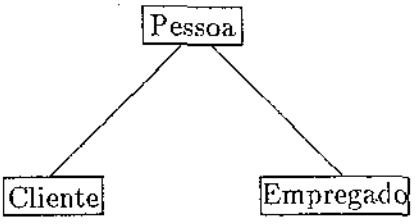
[1] a partir de agora, sempre que for referenciado classe do atributo, significa o mesmo que tipo do atributo

- Tabela T2: classes do esquema, métodos das classes, atributos dos métodos e seus tipos/classes.

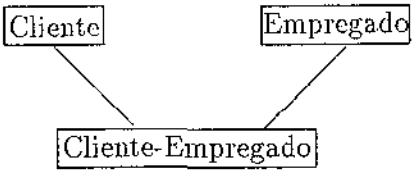
  A partir de T2 pode-se saber os métodos que podem afetar uma determinada restrição quando executados sobre uma classe fonte de violação. Note-se que esta tabela exige informação sobre a execução de um método que não pode ser extraída diretamente do esquema de um banco de dados, por haver dados (atributos acessados dentro do método) que são obtidos apenas em tempo de execução e dependem, assim, da semântica do método. Supõe-se que estes dados sejam inseridos pelo administrador do banco de dados. Desse modo, se dentro de um método **m1** há acesso direto a um atributo x do tipo **t1**, na tabela haverá indicação de que o método m1 modifica x: t1. Por outro lado, se **m1** invoca um método **m2** sobre uma classe **c2**, esta informação não constará na tabela porque corresponde a uma chamada a método e não a modificação direta de atributos.

  A informação "Atributo: tipo" armazenada na tabela T2 identifica os campos que podem ser alterados por cada método e seus respectivos tipos. Outra opção para a tabela seria colocar apenas os parâmetros da assinatura. Se por um lado isso permitiria construção automática da tabela a partir do esquema (pois a assinatura é armazenada no esquema), por outro não permitiria identificar todos os campos atualizados por um determinado método. Um exemplo deste problema é um método sem parâmetros, mas que pode afetar componentes de uma classe. Contratar () é um método que não possui parâmetros, mas que insere um novo objeto empregado na classe Empregado, a partir de informações solicitadas interativamente.

As tabelas T1 e T2 armazenam dados sobre o esquema e os métodos, bem como os atributos afetados por cada método. Para completar a informação sobre o esquema, armazena-se a composição do grafo de herança sob forma de árvore com ponteiros para super e subclasses. O grafo de herança é um grafo que representa a hierarquia entre as classes. Por exemplo, se Pessoa é superclasse das classes Empregado e Cliente, o grafo fica:



No caso de herança múltipla (por exemplo, a classe Cliente-Empregado herda as características das classes Cliente e Empregado), a representação fica:



O grafo de herança permite que as restrições definidas sobre uma superclasse (por exemplo, Empregado) sejam propagadas para suas subclasses (por exemplo, Cliente-Empregado). Usando este grafo, o algoritmo detecta todos os eventos adicionais que devem acionar regras para verificação de restrições herdadas.

A tabela Classe-subclasse representa internamente o grafo de herança do primeiro exemplo.

TABELA Classe-subclasse

Classe	Subclasses
Pessoa	Empregado, Cliente
Cliente	Cliente-Empregado
Empregado	Cliente-Empregado
Cliente-Empregado	–

Descrição do algoritmo

1. Passo1 : Percorrer a restrição escrita na linguagem de restrição proposta e identificar as classes explicitadas cuja atualização pode violar a restrição. No caso da restrição possuir os operadores de pré e pós-condição, o algoritmo termina neste passo. Isso ocorre porque, neste tipo de restrição, o evento é composto do nome do método e da classe que constam na pré ou pós-condição.

2. Passo2 : Identificar os atributos da restrição.

3. Passo3 : **Para cada** atributo identificado no Passo2:

   (a) Pesquisar a tabela T1 para:

      i. Determinar a classe a que pertence o atributo.

      ii. Determinar a classe das características extraídas do atributo.

4. Passo4: **Se** o atributo não for encontrado na tabela T1 (o atributo referenciado por uma classe pode ser herdado de uma superclasse), **então** procurar as superclasses no grafo de herança e repetir o Passo3 para este atributo. **Se** o atributo for encontrado, **então** armazenar a superclasse, a classe, o atributo e a classe a que pertence em uma lista chamada Lclasse-afetada.

5. Passo5 :

   (a) Armazenar as classes encontradas no Passo1 e no Passo3 em uma tabela chamada Tclasse (indica as classes cujos métodos devem ser pesquisados).

   (b) Armazenar os atributos e suas classes encontradas no Passo3 em uma tabela chamada Tclasse-atributo (indica os atributos fontes de violação da restrição).

   (Passos 6 e 7: Identificam os eventos e os colocam na lista Levento.)

6. Passo6 : **Para cada** classe **c** pertencente a Tclasse:

   (a) Fazer acesso à tabela T2, selecionando as classes **c**.

   (b) **Para cada** método **m** da classe **c**:

i. Verificar se pelo menos um dos atributos de **m** e sua classe correspondente está contido na tabela Tclasse-atributo (se o método correspondente pode violar a restrição ao acessar um atributo fonte de violação).

ii. **Se** estiver contido, **então** armazenar a classe **c** e o método **m** na lista Levento.

7. Passo7 : **Para cada** superclasse **sp** e classe **c** pertencente à Lclasse-afetada:

   (a) Fazer acesso à tabela T2, selecionando as classes **sp**.

   (b) **Para cada** método **m** da classe **sp**:

      i. Verificar se pelo menos um dos atributos de **m** e sua classe correspondente está contido na tabela Tclasse-atributo.

      ii. **Se** estiver contido, **então** armazenar a classe **c** e o método **m** na lista Levento.

   (Passos 8 e 9: Determinam o conjunto final de eventos, navegando sobre o grafo de herança.)

8. Passo8 : **Para cada** classe **c** da lista Levento:

   (a) Pesquisar no grafo de herança (tabela Classe-subclasse) as subclasses de **c**.

   (b) Armazenar as subclasses encontradas em uma nova tabela Tclasse e copiar a tabela Tclasse-atributo da classe **c** em uma nova tabela Tclasse-atributo.

   (c) Propagar os eventos correspondentes à classe **c** para suas subclasses.

9. Passo9 : Criar uma nova tabela Tclasse contendo as subclasses das classes de Tclasse e repetir o Passo6, usando esta nova tabela criada e a tabela Tclasse-atributo, que não sofreu modificações. Este passo permite que as subclasses gerem novos eventos, caso possuam métodos que possam violar a restrição.

Os passos do algoritmo estão representados na Figura 4.1.

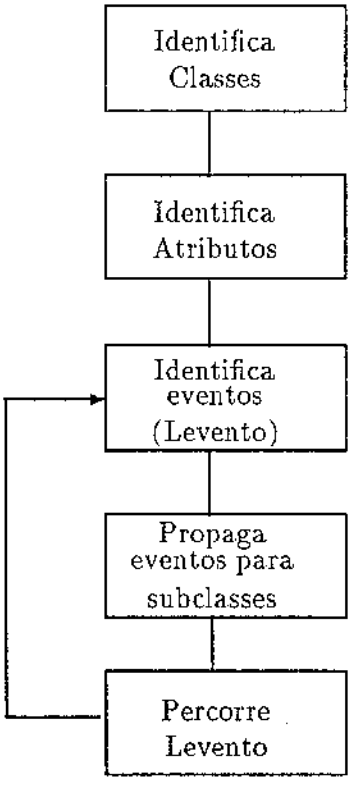


Figura 4.1: Representação dos Passos do Algoritmo

Motivação com Exemplos

Serão dados alguns exemplos de restrições baseados no esquema de dados de uma empresa definido anteriormente, na Figura 3.1. A detecção dos eventos é feita com base no algoritmo já definido. No primeiro exemplo, serão descritos os passos do algoritmo, para propiciar um melhor entendimento.

As informações do esquema dos dados são armazenadas na tabela T1 e T2, conforme as Figuras 4.2 e 4.3 respectivamente.

Classe	Atributo: classe
Pessoa	nome: String, datanasc: Data, endereço: Endereço
Empregado	cargo: Integer, tempo-serviço: Integer, dependente: Pessoa, salário: Dinheiro, dep: Departamento
Cliente	crédito: Dinheiro, status: String
Departamento	nome: String, pessoal: Empregado, gerente: Empregado
Cliente-Empregado	-

Figura 4.2: TABELA T1

Classe	Método	Atributo: classe
Pessoa	atualizar-datanasc	datanasc: Data
Empregado	contratar	Todos atributos de Empregado e Pessoa
Empregado	demitir	Todos atributos de Empregado e Pessoa
Empregado	atualizar-salário	salário: Dinheiro
Empregado	atualizar-dep	dep: Departamento
Departamento	atualizar-gerente	gerente: Empregado pessoal: Empregado
Cliente-Empregado	atualizar-salário	salário: Dinheiro

Figura 4.3: TABELA T2

A representação do grafo de herança fica conforme a Figura 4.4.

Classe	Subclasses
Pessoa	Cliente, Empregado
Cliente	Cliente-Empregado
Empregado	Cliente-Empregado
Cliente-Empregado	–

Figura 4.4: TABELA Classe-subclasse

- Exemplo 1 (vide tabelas na Figura 4.5):

  Restrição interclasse: "Um empregado que é gerente deve ganhar no mínimo 1.000.000".

  (**paratodo** d **em** Departamento; d.gerente.salário >= 1000000)

  Explicação das tabelas T1 e T2:

  Tabela T1: A restrição acima diz respeito às classes Departamento, Empregado e Dinheiro. Para descobrir que a classe Empregado está envolvida nesta restrição, é necessário saber que gerente (atributo da classe Departamento) pertence à classe Empregado. É imprescindível possuir informações sobre os tipos (classes) a que pertencem os atributos de uma determinada classe.

  Tabela T2: Através desta tabela, pode-se descobrir que o método atualizar-salário da classe Empregado pode violar a restrição (pois afeta o campo salário da classe Dinheiro). Isso ocorre porque uma mudança no salário de um empregado implica na necessidade da verificação da restrição, já que este empregado pode ser um gerente. Serão analisados dois métodos da classe Empregado (atualizar-dep e atualizar-salário), a fim de dar uma idéia de como se dá a procura nesta tabela.

  O método atualizar-dep, por exemplo, é descartado, pois não envia mensagem para os atributos salário ou gerente, considerados fontes de violação da restrição. O não envio das mensagens é descoberto por uma verificação nos atributos do método em questão. Como o atributo do método atualizar-dep é diferente de salário e gerente, então o método não é selecionado. Seguindo o mesmo raciocínio feito para o método atualizar-dep, constata-se que o método atualizar-salário deve ser selecionado, por possuir pelo menos um atributo considerado fonte

de violação de restrição (no caso, o atributo salário).

Passos do algoritmo:

1. Passo1 : Classes identificadas: Departamento.
2. Passo2 : Atributos identificados: d.gerente e d.gerente.salário.
3. Passo3 : d.gerente pertence à classe Empregado e d.gerente.salário pertence à classe Dinheiro.
4. Passo4 : Não repete o Passo3, pois o atributo já foi encontrado na tabela T1.
5. Passo5 : Tclasse contém Departamento, Empregado e Dinheiro. Tclasse-atributo contém gerente: Empregado e salário: Dinheiro.
6. Passo6 : Análise da classe Departamento:

   (a) Método atualizar-gerente - selecionado (um dos atributos do método é igual a gerente: Empregado).

   Análise da classe Empregado:

   (a) Método contratar - selecionado (um atributo do método é igual a salário: Dinheiro).
   (b) Método demitir - selecionado (um atributo do método é igual a salário: Dinheiro).
   (c) Método atualizar-salário - selecionado (o atributo do método é igual a salário: Dinheiro).
   (d) Método atualizar-dep - descartado (o atributo do método difere de gerente: Empregado e salário: Dinheiro).
7. Passo7 : Não existe a lista Lclasse-afetada.
8. Passo8 : Repetir o Passo6 para a subclasse Cliente-Empregado da classe Empregado armazenada na tabela Tclasse (a única subclasse envolvida é Cliente-Empregado, pois Departamento e Dinheiro não possuem subclasses). Neste passo é feita a propagação dos eventos para as subclasses.
9. Passo6 : Análise da classe Cliente-Empregado:

   (a) Método atualizar-salário - selecionado (o atributo do método é igual a salário: Dinheiro). Não precisa inserir em Levento, porque já foi inserido durante a propagação dos eventos (Passo8).

10. Passo7 : Não existe a lista Lclasse-afetada.
11. Passo8 : Cliente-Empregado não possui subclasses.
12. Passo9 : Não houve mudanças em Tclasse, porque Cliente-Empregado não possui subclasses.
13. Resultado: os eventos que podem violar a restrição:
    - <Departamento, atualizar-gerente>
    - <Empregado, atualizar-salário>
    - <Empregado, contratar>
    - <Empregado, demitir>
    - <Cliente-Empregado, atualizar-salário>
    - <Cliente-Empregado, contratar>
    - <Cliente-Empregado, demitir>

- Exemplo 2 (Tabelas na Figura 4.6):

  Restrição intraclasse: "O salário de um gerente é maior que o salário de todos os empregados do departamento".

  (**paratodo** x **em** Empregado; x.salário <= x.dep.gerente.salário)

- Exemplo 3 (Tabelas na Figura 4.7):

  Restrição intraclasse: "A idade de uma pessoa cadastrada é maior que 25 anos ".

  (**paratodo** p **em** Pessoa; p.datanasc < 230791[2])

  Obs.: Nesta restrição ocorre a propagação do evento para as subclasses Empregado e Cliente. O Passo9 do algoritmo é executado e o passo6 é repetido para as tabelas criadas (Tclasse e Tclasse-atributo), Tclasse contendo Empregado e Cliente e Tclasse-atributo contendo datanasc: Data. Note-se que, além disso, são detectados eventos relativos a contratar e demitir empregados. Ambos foram incluídos porque fazem acesso ao atributo datanasc: Data, embora se trate de subclasse de Pessoa, ou seja, houve herança de restrição, inclusive, com métodos que não afetam Pessoa. Demitir foi incluído, apesar de não violar a restrição. Este fato, no entanto, só poderia ser verificado com auxílio do usuário, pois as informações utilizadas pelo algoritmo não são suficientemente precisas para determinar este tipo de diferença semântica.

[2]considere que os campos do tipo data possuem o formato ddmmaa

- Exemplo 4 (Tabelas na Figura 4.8):

  Restrição intraclasse: "A idade de um empregado é maior que 25 anos ".

  (**paratodo** x **em** Empregado; x.datanasc < 230791)

  Obs.: Nesta restrição, o atributo datanasc, referenciado pela classe Empregado, pertence a sua superclasse Pessoa. Logo, os métodos pesquisados são os da classe Pessoa. O método encontrado é propagado para a classe Empregado, que, por sua vez, propaga para sua subclasse Cliente-Empregado. Note-se que, embora Cliente seja subclasse de Pessoa, o algoritmo ignora o fato, já que se trata de identificar eventos referentes a Empregado (pela restrição) e subclasses de Empregado (herança múltipla).

- Exemplo 5 (Tabelas na Figura 4.9):

  Restrição intraclasse: "O gerente de um departamento pertence ao quadro de pessoal do seu departamento ".

  (**paratodo** d **em** Departamento; **estacontido**(d.gerente, d.pessoal))

- Exemplo 6 (Tabelas na Figura 4.10):

  Restrição interclasse: "O salário do empregado não pode exceder duas vezes o máximo do salário dos empregados do seu departamento".

  (**paratodo** x **em** Empregado; x.salário <=

  2*max(x.dep.pessoal.salário))

- Exemplo 7 (Tabelas na Figura 4.11):

  Restrição interclasse: "O somatório dos salários dos empregados de um departamento é superior a 100.000".

  (**paratodo** d **em** Departamento; sum(d.pessoal.salário) >= 100000)

4.3.4 Aplicação em Sistemas não OO

O sistema proposto serve também para qualquer banco de dados relacional ou aninhado. Neste caso, não há muitos métodos, apenas as operações *insert*, *update* e *delete*. Não há tampouco superclasses ou subclasses, e as tabelas T1 e T2 possuem no lugar do campo classe o campo relação e no lugar dos campos atributos:tipo, permanece o campo atributo:tipo, quando se trata de banco de dados relacional) ou substitui-se por atributo:relação (no caso

de aninhamento é colocado o nome da relação que compõe o atributo). A tabela Tclasse passa a conter os nomes das relações envolvidas na restrição e a tabela Tclasse-atributo passa a conter os nomes das relações e os nomes dos atributos afetados. Com estas simplificações, o algoritmo generaliza o proposto por [CW90], para sistemas relacionais.

O processamento do algoritmo não é alterado, apenas percorre menos tabelas. Logo, o algoritmo é considerado de uso geral, uma vez que pode ser utilizado para sistemas OO, relacionais e relacionais estendidos.

Imagine a seguinte restrição para um banco de dados relacional, utilizando a relação Empregado (nome, num-matr, salário, num-dep): "o salário de um empregado não pode exceder duas vezes o salário médio do seu departamento". A consulta feita ao banco de dados para extrair as tuplas que violam esta restrição, baseado em [CW90], fica:

```
if exists (select * from Empregado e1
           where salario > 2*
                 (select avg(salario)
                  from Empregado e2
                  where e2.num-dep =
                        e1.num-dep)
```

A relação detectada nesta restrição é Empregado e os atributos são salário e num-dep. Sendo assim, as tabelas T1, T2, Tclasse, Tclasse-atributo e Levento, referentes a esta restrição, podem ser visualizadas na Figura 4.12. Observe que as tabelas T1 e T2 dizem respeito apenas às relações base do esquema do banco de dados e não às visões.

Note-se que Tclasse contém o nome da relação envolvida na restrição e Tclasse-atributo contém o nome do atributo referenciado e seu respectivo tipo. Os eventos detectados a partir de uma análise dos métodos estão na tabela Levento.

[WF90] utiliza esse exemplo de restrição. As operações detectadas que a invalidam são exatamente as mesmas mostradas na lista de eventos da Figura 4.12.

Conclui-se que, a partir do exemplo considerado, a utilização do algoritmo para bancos de dados relacionais é trivial.

4.3.5 Criação da Regra

Utilizando o algoritmo anterior, as regras $< \mathcal{E}, \mathcal{C}, \mathcal{A} >$ são obtidas da seguinte forma. Seja uma restrição $\mathcal{R}$ expressa como um predicado $\mathcal{P}$, e a ação

correspondente $\mathcal{A}$, opcional. Esta restrição é analisada para determinar as regras correspondentes, que terão todas os mesmos componentes para $\mathcal{C}$ e $\mathcal{A}$. A expressão correspondente à restrição é

$\neg\mathcal{P} \rightarrow \mathcal{A}$

(ou seja, se o predicado não é obedecido, a ação deve ser executada). Quando a ação é omitida, supõe-se a ação-padrão (prevenção), quer dizer, se o predicado não é obedecido, a operação correspondente ao evento é rejeitada.

O componente $\mathcal{A}$ das regras é obtido diretamente da expressão. O predicado $\mathcal{P}$ dá origem aos dois outros componentes:

- a condição $\mathcal{C} = \neg\mathcal{P}$;
- o conjunto de eventos, a partir da análise de $\mathcal{R}$, seguindo o algoritmo deste capítulo.

Tclasse

Departamento
Empregado
Dinheiro

Tclasse-atributo

gerente: Empregado
salário: Dinheiro

Levento inicial

Classe	Método
Departamento	atualizar-gerente
Empregado	atualizar-salário
Empregado	contratar
Empregado	demitir

Levento final

Classe	Método
Departamento	atualizar-gerente
Empregado	atualizar-salário
Empregado	contratar
Empregado	demitir
Cliente-Empregado	atualizar-salário
Cliente-Empregado	contratar
Cliente-Empregado	demitir

Figura 4.5: Exemplo 1

Tclasse

Empregado
Dinheiro
Departamento

Tclasse-atributo

dep: Departamento
gerente: Empregado
salário: Dinheiro

Levento

Classe	Método
Empregado	contratar
Empregado	demitir
Empregado	atualizar-salário
Empregado	atualizar-dep
Departamento	atualizar-gerente
Cliente-Empregado	contratar
Cliente-Empregado	demitir
Cliente-Empregado	atualizar-salário
Cliente-Empregado	atualizar-dep

Figura 4.6: Exemplo 2

Tclasse

Pessoa
Data

Tclasse-atributo

datanasc: Data

Levento

Classe	Método
Pessoa	atualizar-datanasc
Empregado	atualizar-datanasc
Cliente	atualizar-datanasc
Cliente-Empregado	atualizar-datanasc
Empregado	contratar
Empregado	demitir
Cliente-Empregado	contratar
Cliente-Empregado	demitir

Figura 4.7: Exemplo 3

Tclasse

Empregado

Tclasse-atributo

datanasc: Data

Lclasse-afetada

Superclasse	Classe	Atributo: Tipo
Pessoa	Empregado	datanasc: Data

Levento

Classe	Método
Empregado	contratar
Empregado	demitir
Empregado	atualizar-datanasc
Cliente-Empregado	contratar
Cliente-Empregado	demitir
Cliente-Empregado	atualizar-datanasc

Figura 4.8: Exemplo 4

Tclasse

Departamento
Empregado

Tclasse-atributo

gerente: Empregado
pessoal: Empregado

Levento

Classe	Método
Departamento	atualizar-gerente

Figura 4.9: Exemplo 5

Tclasse

Empregado
Departamento
Dinheiro

Tclasse-atributo

dep: Departamento
pessoal: Empregado
salário: Dinheiro

Levento

Classe	Método
Empregado	atualizar-salário
Empregado	contratar
Empregado	demitir
Empregado	atualizar-dep
Departamento	atualizar-gerente
Cliente-Empregado	atualizar-salário
Cliente-Empregado	contratar
Cliente-Empregado	demitir
Cliente-Empregado	atualizar-dep

Figura 4.10: Exemplo 6

Tclasse

Departamento
Empregado
Dinheiro

Tclasse-atributo

pessoal: Empregado
salário: Dinheiro

Levento

Classe	Método
Departamento	Atualizar-gerente
Empregado	atualizar-salário
Empregado	demitir
Empregado	contratar
Cliente-Empregado	atualizar-salário
Cliente-Empregado	demitir
Cliente-Empregado	contratar

Figura 4.11: Exemplo 7

Tabela T1

Relação	Atributo: tipo
Empregado	nome: String, num-matr: Integer salário: Integer, num-dep: Integer

Tabela T2

Relação	Operação	Atributo: tipo
Empregado	*insert*	todos atributos de Empregado
Empregado	*delete*	todos atributos de Empregado
Empregado	*update (salário)*	salário: Integer
Empregado	*update (dep)*	num-dep: Integer
Empregado	*update (nome)*	nome: String

Tclasse

Empregado

Tclasse-atributo

salário: Integer num-dep: Integer

Levento

Relação	Operação
Empregado	*insert*
Empregado	*delete*
Empregado	*update (salário)*
Empregado	*update (dep)*

Figura 4.12: Exemplo para um Banco de Dados Relacional

Capítulo 5

Implementação

5.1 Introdução

A implementação feita para o algoritmo do capítulo anterior será apresentada neste capítulo. Primeiramente serão feitas algumas considerações sobre o sistema desenvolvido. Em seguida, será dada uma visão geral do sistema. Finalmente, será detalhado o funcionamento dos principais módulos do sistema e serão mostradas as estruturas de dados utilizadas. O desenvolvimento deste sistema teve por fim validar algumas das idéias expostas no decorrer da tese.

5.2 Considerações Iniciais

O algoritmo de restrições foi implementado em C na rede de estações SPARC do DCC UNICAMP.

O sistema desenvolvido visa servir como uma camada de suporte às restrições definidas para um banco de dados orientado a objetos. Esta camada pressupõe que o SGBD subjacente possui um mecanismo de regras (como, por exemplo, implementado no sistema O_2 [MP91a] ou Damascus [KDM88]). As restrições especificadas são analisadas, e os eventos são detectados segundo o algoritmo do capítulo 4. É gerada uma regra para cada evento usando a idéia de que cada restrição corresponde a uma condição (predicado) e a uma ação, além de um conjunto de eventos. Os componentes (eventos, condição, ação) são passados para o SGBD, para que este gere as regras segundo o mecanismo já implementado.

A camada implementada utiliza duas fontes de dados: restrição de in-

tegridade, fornecida pelo usuário na linguagem proposta no capítulo 4, e informações sobre o esquema. Estas últimas são lidas de um arquivo intermediário, de formato-padrão. A criação deste arquivo exige a intervenção de um módulo que acesse o banco de dados para obter as informações necessárias do esquema (através da DML do SGBD específico) e que obtenha do projetista informações sobre a semântica dos métodos (para permitir definir a tabela T2). Este módulo, variável a cada SGBD, cria os dados correspondentes às tabelas (T1 e T2) descritas no capítulo 4. Assim, a cada SGBD corresponde um módulo próprio.

A nível conceitual, uma grande vantagem do sistema desenvolvido é que pode ser acoplado a qualquer banco de dados orientado a objetos, pois sua implementação não foi baseada em um SGBDOO específico. O sistema utiliza informações do esquema de dados a partir de um modelo que, de acordo com [ABD+89], qualquer modelo orientado a objetos deve possuir. Além disso, a escolha da linguagem C contribui para que o sistema seja utilizado em ambientes diferentes daquele em que foi projetado.

O acoplamento do sistema implementado a um SGBD qualquer exige a existência dos seguintes módulos já incorporados ao SGBD:

- módulo de captação de intormações do esquema para criação das tabelas T1 e T2;
- módulo de criação e manutenção de regras, definidas como <E, C, A>, que seriam fornecidas ao sistema pela camada desenvolvida nesta tese.

Naturalmente, soluções mais específicas voltadas para peculiaridades de cada SGBD podem também ser implementadas usando os conceitos discutidos na tese. Isto exigiria, no entanto, conhecimento específico de cada SGBD, para assim desenvolver sistemas de melhor desempenho e melhor adaptados para cada caso.

5.3 Visão Geral

A estrutura do sistema implementado está representada na Figura 5.1. O módulo de captação de informações do esquema (CARREGA CLASSES, ATRIBUTOS E MÉTODOS) é responsável pela leitura do arquivo-padrão contendo o esquema do banco de dados. Os dados (sobre classes, atributos e métodos) são colocados em estruturas de dados intermediárias, acessados pelo módulo de geração de eventos.

A cada restrição fornecida, é feita uma análise sintática (módulo ANÁLISE SINTÁTICA) descendente recursiva da restrição escrita na linguagem de restrição.

O módulo seguinte (IDENTIFICA VARIÁVEIS, ATRIBUTOS e CLASSES) é responsável pela separação dos atributos e classes referenciados na restrição e não precisa se preocupar em analisar a sintaxe da restrição, pois qualquer erro é detectado pelo módulo anterior e comunicado ao usuário, para que haja a devida correção.

Conforme definido no capítulo 4, os eventos detectados são pares <classe, método> que ocasionam a verificação da restrição. A geração de tais eventos (módulo GERA EVENTOS) constitui o passo fundamental do sistema desenvolvido. Este módulo utiliza as informações da restrição e consulta as informações do esquema nas estruturas intermediárias para gerar os eventos a serem verificados.

5.4 Detalhamento dos Principais Módulos

Serão mostrados em detalhes alguns módulos do sistema, buscando, assim, propiciar um melhor entendimento do mesmo. Os algoritmos para estes módulos encontram-se descritos em pseudocódigo, no apêndice B.

Módulo: CARREGA CLASSES, ATRIBUTOS E MÉTODOS

Este módulo tem por função ler as informações do esquema de dados (já obtidas a partir do módulo de captação de informação específico ao SGBD) e carregá-las em estruturas de dados de fácil manipulação.

As informações do esquema são de três tipos: informações sobre composição e tipos de atributos, informações sobre métodos e informações sobre herança. O algoritmo inicialmente cria estruturas correspondentes à composição. As classes do esquema são armazenadas em uma árvore AVL, onde cada nó contém o nome da classe e ponteiros para os métodos e os atributos componentes. Os atributos de cada classe, por sua vez, são também armazenados em uma árvore AVL separada, onde cada nó contém o nome do atributo e a classe (tipo) respectiva.

Os métodos são armazenados em uma lista ligada. Cada elemento da lista contém o nome do método e ponteiros para os atributos correspondentes (ou seja, listas ligadas de atributos).

Em outras palavras, há pelo menos duas estruturas distintas para des-

crição dos atributos: a árvore referenciada pelo grafo de classes e a lista referenciada pela lista de métodos, ou seja, o método, ao referenciar um atributo de uma classe, não se aproveita do fato desta informação já se encontrar armazenada. Assim, cria sua própria lista de atributos. Se vários métodos referenciam o mesmo atributo, este se repetirá em cada lista criada por cada método. Ainda que esta opção não seja ideal, em termos de economia de espaço, foi adotada para simplificar a implementação. Caso contrário, seria necessário testar, a cada método da lista, a igualdade de seus atributos com a de outros métodos já inseridos.

Esse teste de igualdade é demorado. Para efeito da criação da lista de métodos, significa verificar não apenas se dois atributos têm o mesmo nome, mas se têm o mesmo tipo e compõem a mesma classe. Isso ocorre porque o algoritmo que estabelece os eventos <Classe, Método> usando informações do esquema adiciona à lista de eventos um determinado evento <C, M> apenas se o método M fizer acesso a algum atributo fonte de violação de restrição. Sendo assim, é necessário que as informações sobre os atributos incluam o nome, o tipo e a classe a que pertence o atributo. Com o teste de igualdade, haveria, em princípio, apenas uma estrutura para cada atributo, referenciada pelo grafo de classes e pelos métodos correspondentes.

Como a implementação pretende apenas validar o algoritmo, e esta última opção aumentaria a complexidade do código sem ganho em resultados desejados (determinação da regra), optou-se pela primeira alternativa. Da mesma forma, a implementação em árvores AVLs para várias estruturas, visou aliar simplicidade de algoritmos à rapidez de busca de estruturas em memória.

Para a representação dos métodos das classes, optou-se por uma lista simplesmente encadeada. O motivo para esta escolha é que o tempo de busca não é relevante, visto que a lista será percorrida uma única vez, do início ao fim, pelo módulo que necessita analisar todos os métodos da classe, não havendo possibilidade de otimizar a pesquisa. A Figura 5.2 mostra a estrutura do grafo de composição e dos métodos.

O grafo de herança que reflete os relacionamentos entre as superclasses e subclasses do esquema de dados é representado pelas estruturas de dados da Figura 5.3. Optou-se por duas representações, das superclasses e das subclasses, pelo fato de que a navegação do grafo de herança se torna, desse modo, mais fácil. No caso da representação das superclasses, as classes que estão contidas no grafo são inseridas em uma árvore AVL. Cada nó desta árvore possui um apontador para uma lista de adjacência de suas superclasses. A representação das subclasses segue o mesmo raciocínio da repre-

sentação das superclasses. A diferença é que a lista de adjacência contém as subclasses da classe e não as superclasses. Além da vantagem de visualização obtida com as duas representações utilizadas para o grafo de herança, uma outra vantagem é a facilidade de remoção e atualização das classes que compõem o grafo resultante do uso destas estruturas.

Vale ressaltar que, para a escolha das estruturas de dados, o tempo de pesquisa foi o critério considerado mais importante. O tempo de inserção, apesar de relevante, não afeta muito o processamento do algoritmo, pois a inserção das informações do esquema ocorre apenas uma vez para o conjunto de restrições relativas àquele esquema de dados. O tempo gasto para remoção e atualização das informações nestas estruturas não é significativo porque estas operações não ocorrem com muita freqüência durante o processamento das restrições. A possibilidade de modificação do esquema em paralelo à criação de restrições faz parte do conjunto de extensões à dissertação, descritos no capítulo 6.

Para melhor exemplificar o armazenamento dos dados do esquema nas estruturas propostas, será mostrado na Figura 5.4 a representação do banco de dados de uma empresa (descrito anteriormente). A classe Empregado é a escolhida para ser detalhada. Na lista de métodos não aparecem todos os métodos da classe Empregado, apenas atualizar-salário e contratar. É mostrada apenas a lista de atributos do método atualizar-salário. Apenas uma porção do banco de dados é representada nesta figura, mas buscou-se, através deste exemplo, dar uma idéia geral da estrutura empregada.

Módulo: IDENTIFICA VARIÁVEIS, ATRIBUTOS E CLASSES

Neste módulo é lida cada palavra que compõe a restrição. Para cada palavra lida, é testado se a palavra é uma variável, uma classe ou um atributo. A partir da gramática definida no capítulo 4, sabe-se que uma variável aparece após os quantificadores existenciais e/ou universais, que uma classe sempre vem após a palavra reservada "em" e que os atributos contêm um símbolo especial (".") entre os identificadores. Como o módulo anterior a este (ANÁLISE SINTÁTICA) garante a correção sintática da restrição, o teste sobre a palavra lida para a detecção das variáveis, classes e atributos torna-se extremamente simples com o uso das informações de localização extraídas da gramática da linguagem de restrição.

Por fim, são gerados o conjunto das variáveis e suas respectivas classes e o conjunto dos atributos da restrição que serão utilizados pelo módulo seguinte.

Módulo: GERA EVENTOS

Este módulo gera os eventos responsáveis pela verificação da restrição. Isso é feito a partir de informações sobre a restrição e sobre o esquema de dados, sendo as informações específicas de cada restrição armazenadas em estruturas adicionais às do esquema: a cada restrição são eliminadas as estruturas da restrição anterior e criadas novas estruturas relativas à restrição corrente.

O conjunto dos atributos da restrição obtidos no módulo IDENTIFICA VARIÁVEIS, ATRIBUTOS E CLASSES é percorrido e, a cada característica extraída de um atributo da restrição (vide capítulo 4), é buscado seu tipo correspondente nas estruturas de esquema geradas no módulo CARREGA CLASSES, ATRIBUTOS E MÉTODOS. Desta forma, determina-se os tipos dos atributos referenciados pela restrição.

Com isso são geradas as informações para TCLASSE e TCLASSE-ATRIBUTO descritas no capítulo 4, que são então armazenadas como árvores AVL. TCLASSE contém as classes das variáveis, dos atributos e de suas características extraídas, enquanto que TCLASSE-ATRIBUTO contém apenas as classes dos atributos e de suas características extraídas.

Finalmente, os eventos são gerados e armazenados em uma lista LISTA-EVENTO. A determinação dos eventos é feita percorrendo primeiro TCLASSE e depois LCLASSE-AFETADA (lista de superclasses). Para cada classe de TCLASSE, verifica-se se seus métodos têm ao menos um atributo que está contido em TCLASSE-ATRIBUTO. Caso afirmativo, é gerado o evento correspondente, formado pela classe em TCLASSE e pelo método detectado. Em seguida, para cada superclasse em LCLASSE-AFETADA, realiza-se a mesma verificação quanto aos métodos. No caso desses métodos também gerarem violação, darão origem à inserção de eventos adicionais em LISTA-EVENTO.

Será descrito, a seguir, um exemplo de tratamento de restrições para facilitar o entendimento. Este exemplo é baseado no esquema de dados de uma empresa utilizado nos capítulos anteriores. Imagine a restrição: (**paratodo** x **em** Empregado; x.datanasc > 260167). O algoritmo precisa saber a classe do atributo x.datanasc referenciado na restrição. A palavra **em** indica que x é uma variável da classe Empregado. Esta classe é pesquisada na árvore de classes do esquema. Encontrada a classe, o atributo datanasc é pesquisado na árvore de atributos da classe Empregado. No caso, houve insucesso da pesquisa, e pressupõe-se que a classe Empregado esteja referenciando-se a um atributo herdado. Sendo assim, é utilizada a representação do grafo de herança para que sejam descobertas as superclasses da classe Empregado.

Detecta-se, então, que a classe Pessoa é uma superclasse de Empregado e repete-se o processo de busca do atributo, agora na classe Pessoa. Neste caso, o atributo é encontrado.

Em resumo, a pesquisa nas estruturas do esquema da classe de um atributo e de suas características extraídas é feita, inicialmente, usando a classe que o referencia diretamente. No caso de insucesso, é feita uma pesquisa recursiva com as superclasses desta classe. Se o atributo for detectado em alguma superclasse, esta informação é registrada em uma lista simplesmente encadeada correspondente a LCLASSE-AFETADA mostrada no capítulo 4. Esta lista conteria a superclasse Pessoa, onde o atributo foi encontrado; a classe Empregado, referenciada na restrição; e, o atributo datanasc e sua respectiva classe, no caso Data. Sendo assim, no exemplo da restrição acima, o método atualizar-datanasc, definido na classe Pessoa, seria selecionado, e o evento <Empregado, atualizar-datanasc> é que seria inserido na lista de eventos e não o evento <Pessoa, atualizar-datanasc>. Isso acontece porque a restrição diz respeito à classe Empregado e não à classe Pessoa.

Por fim, é feita a propagação dos eventos para as subclasses das classes que se encontram na LISTA-EVENTO com a geração de novos eventos pelas subclasses afetadas como conseqüência desta propagação.

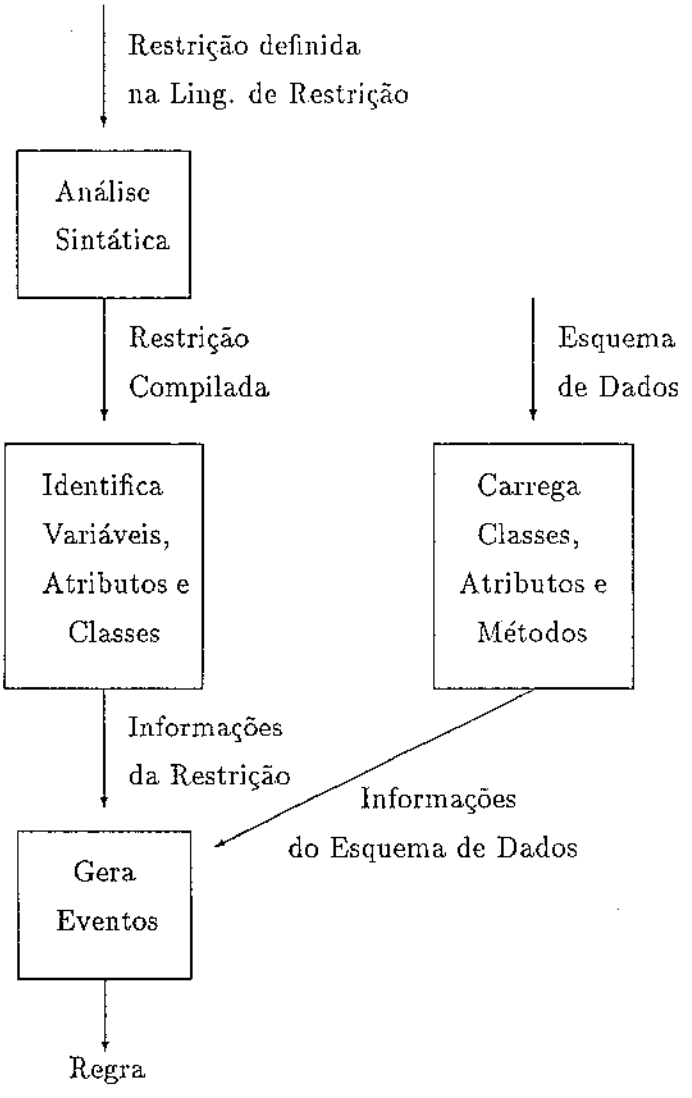


Figura 5.1: Estrutura do Sistema de Transformação de Restrições em Regras

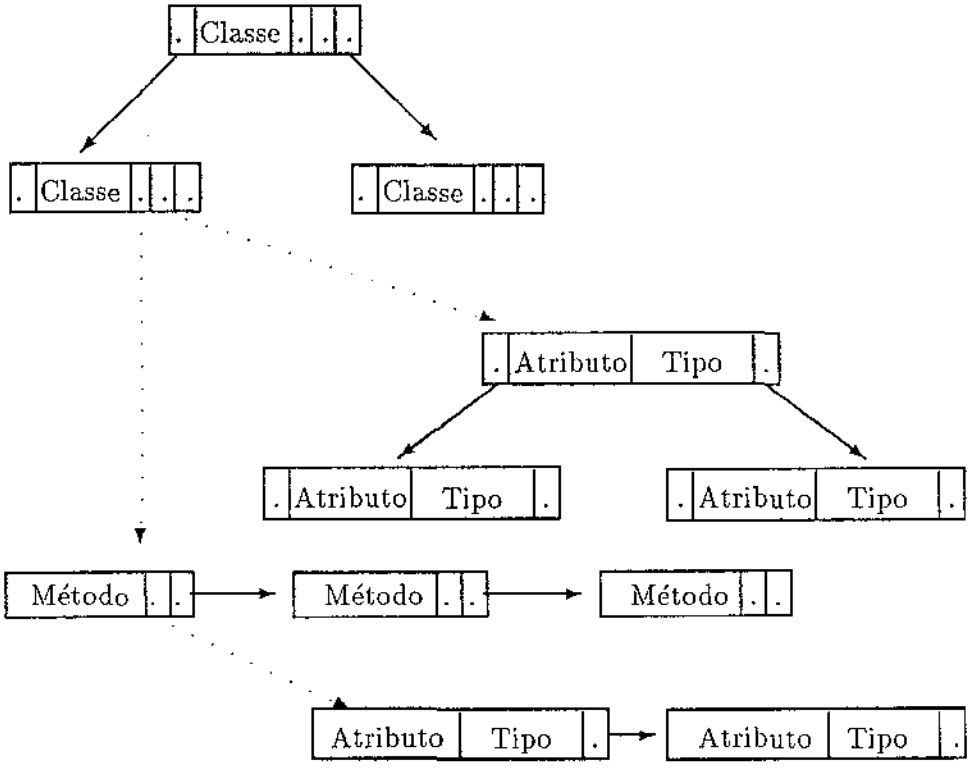


Figura 5.2: Representação do Esquema de Dados

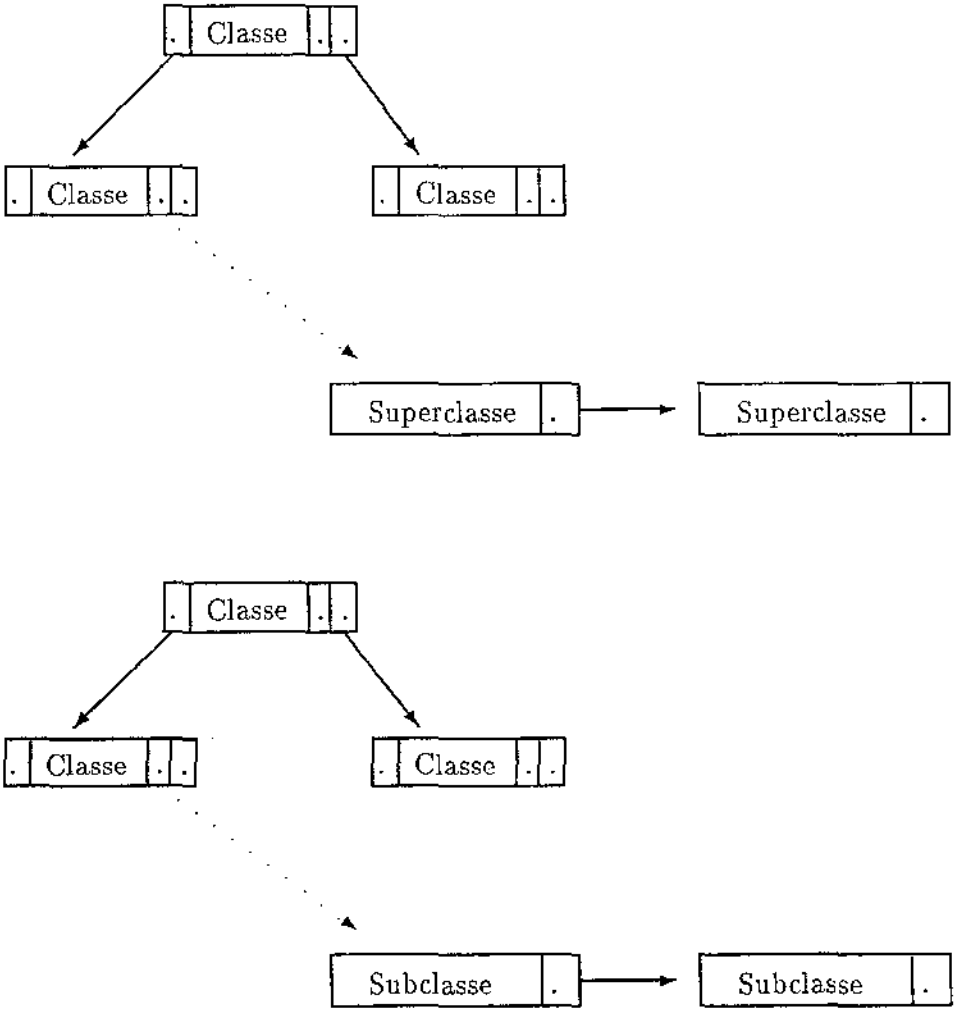


Figura 5.3: Representação do Grafo de Herança

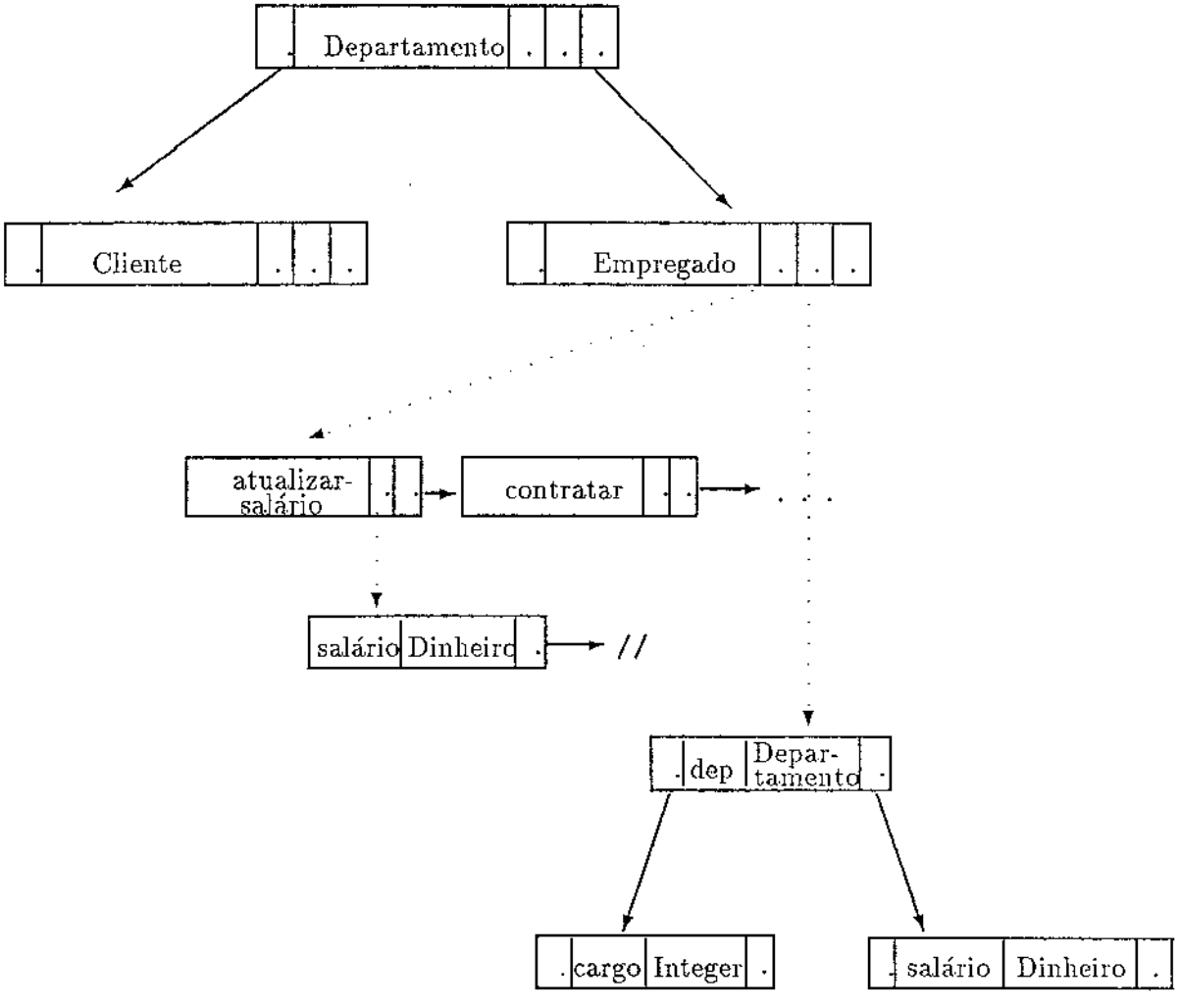


Figura 5.4: Representação do Esquema de Dados de uma Empresa

Capítulo 6

Conclusão

Esta dissertação discutiu o problema da manutenção de restrições estáticas em sistemas OO e apresentou uma solução para o tratamento destas restrições utilizando o paradigma de regras de produção.

Inicialmente, no capítulo 1, foi mostrado o assunto para abordagem: restrições e banco de dados orientado a objetos. Procurou-se dar uma visão geral do problema, destacando conceitos básicos necessários no decorrer da dissertação.

No capítulo 2, foi feita uma revisão bibliográfica sobre tratamento de restrições em sistemas OO. Os trabalhos apresentados foram os mais significativos da área, sendo que a maior parte dos pesquisadores propôs a transformação de restrições em regras para um melhor tratamento das restrições.

Em seguida, no capítulo 3, propôs-se uma taxonomia de restrições estáticas no modelo OO para melhor caracterizar as restrições utilizadas neste trabalho.

O capítulo 4 apresentou um algoritmo para transformação automática de uma restrição de integridade em um conjunto de regras de produção. A parte principal do algoritmo é a detecção dos eventos responsáveis pela verificação da restrição. Foi proposta, ainda, uma linguagem de definição de restrições baseada em lógica de primeira ordem, sem negação.

Finalmente, no capítulo 5, foi descrita a implementação do algoritmo mostrado anteriormente com apresentação das estruturas de dados utilizadas. O sistema apresentado generaliza trabalhos anteriores na área de verificação de restrições através do paradigma de bancos de dados ativos.

Dentre as principais contribuições deste trabalho, pode-se citar:

- discussão do problema de manutenção de integridade em sistemas OO, com implementação (ainda que experimental) de uma solução. Ressalte-se que a maioria dos trabalhos existentes não provê a transformação automática de restrições em regras.
- o sistema desenvolvido pode ser utilizado para gerar regras para vários tipos de sistemas de banco de dados: orientados a objetos, relacionais e relacionais aninhados.
- o sistema provê tratamento de restrições não apenas sobre dados, mas também sobre métodos.
- finalmente, como parte integrante do trabalho, foi desenvolvida uma taxonomia para restrições em sistemas OO, que considera sua dimensão dinâmica, sendo proposta uma linguagem de especificação de restrições.

6.1 Extensões

Esta seção aponta algumas extensões para o trabalho desenvolvido. Uma delas seria a utilização de informações extraídas dos operadores de agregação e seu uso em restrições sobre campos agregados. Outra seria a manutenção automática de restrições quando ocorrem modificações no esquema de dados (ou seja, evolução das restrições ao longo do tempo). Ainda outra seria a utilização de estruturas mais eficientes quanto ao armazenamento de dados. As extensões encontram-se detalhadas nas subseções seguintes.

6.1.1 Extensão do Algoritmo para Operadores de Agregação

Alguns eventos gerados pelo algoritmo podem ser descartados se for criado um campo Tipo-atualização na tabela T2, associado a cada atributo do método. A importância desta informação pode ser vista pelo exemplo dado a seguir. Imagine a restrição: "O somatório dos salários dos empregados de um departamento é maior que 1.000.000". A princípio, pode-se pensar que um método atualizar-salário, quando executado, afetaria a restrição. Contudo, se o método atualizar-salário executasse sempre aumento de salário e nunca o contrário, este método nunca violaria a restrição. A informação que constaria na tabela T2, no campo Tipo-atualização, seria M+, indicando modificação para um valor superior.

A única atualização que afetaria essa restrição seria, para ilustrar, a demissão de um empregado (método demitir), porque o somatório dos salários sofreria um decréscimo se tal método fosse ativado, o que poderia levar à violação da restrição. Neste caso, seria possível tirar vantagem de operadores de agregação (sum, max, min) da linguagem e possivelmente aumentá-la com outros operadores.

Uma possível solução para saber quais operações afetam um agregado consiste em criar uma tabela adicional. Sendo assim, nela se incluiriam os operadores de agregação junto com os possíveis operadores de comparação e os tipos de alteração que têm influência sobre eles. Então, a alteração que influencia "sum(x)" é a exclusão e/ou alteração para um valor inferior de objetos pertencentes a x, ou seja, alterações que façam o valor do somatório decrescer, implicando, assim, na necessidade de verificação da restrição. A interpretação para os outros operadores da tabela seria feita de modo similar. A possível tabela dos operadores de agregação está representada na Figura 6.1, onde as letras I, E, M indicam inclusão, exclusão e modificação. M+ e M− correspondem ao aumento e à diminuição de valores.

Operador de agregação	Comparador	Atual-op
sum	$>\geq$	EM-
sum	$<\leq$	IM+
max	$>\geq$	IEM-
max	$<\leq$	IEM+
min	$>\geq$	IEM-
min	$<\leq$	IEM+

Figura 6.1: TABELA dos Operadores de Agregação

6.1.2 Extensão do Algoritmo para Manutenção Automática de Restrições

Note que o algoritmo pressupõe leitura de informações do esquema. O módulo de captação destas informações deve se encarregar de manter os arquivos intermediários atualizados, quando há modificações no esquema do banco de dados.

No algoritmo implementado, uma modificação no banco de dados requer que as restrições definidas sejam revistas para possível modificação do con-

junto de regras. Uma extensão possível consiste em desenvolver um sistema que garanta que, quando o esquema for atualizado, as regras relativas às restrições vigentes sejam mantidas e que as regras que não tenham mais validade, como conseqüência das modificações no esquema, sejam removidas de forma automática.

Uma solução para o problema é manter o texto das restrições já analisadas em um arquivo, com indicação das regras correspondentes, que deve ser consultado a cada mudança do esquema. Esta opção pode não ser adequada se as mudanças forem freqüentes. Uma possibilidade de melhoria é manter associação entre as regras e as restrições que as geraram, criando estruturas armazenadas no banco de dados para este fim. Um exemplo seria criar classes de regras onde cada uma estaria ligada a uma restrição. Uma vantagem adicional neste caso desta seria a de permitir classificar regras segundo a aplicabilidade. Apresenta, por outro lado, o problema de sobrecarregar o sistema com manutenção de associações adicionais.

6.1.3 Implementação Mais Eficiente do Algoritmo com Economia de Estruturas Usadas

Todas as estruturas do esquema poderiam ser transformadas em um único grafo, onde cada nó conteria o nome de uma classe e apontaria para a respectiva árvore de atributos e lista de métodos. A lista de métodos, por sua vez, conteria listas de ponteiros para nós das árvores de atributos correspondentes (da classe e de outras classes).

O grafo de herança seria igualmente representado no mesmo grafo, através de novas arestas indicando direção de herança. O grande problema, neste caso, seria realizar buscas, tendo em vista o grande número de ponteiros e todos os caminhos que deveriam ser percorridos. A Figura 6.2 dá uma idéia de como ficariam os dados de uma empresa armazenados na nova estrutura proposta.

6.1.4 Outras Extensões

A garantia de restrições de integridade em sistemas orientados a objetos apresenta muitos problemas em aberto. Algumas outras extensões possíveis ao trabalho desenvolvido seriam:

- análise do problema de manutenção de restrições dinâmicas;
- manutenção de restrições na presença de versões de objetos;

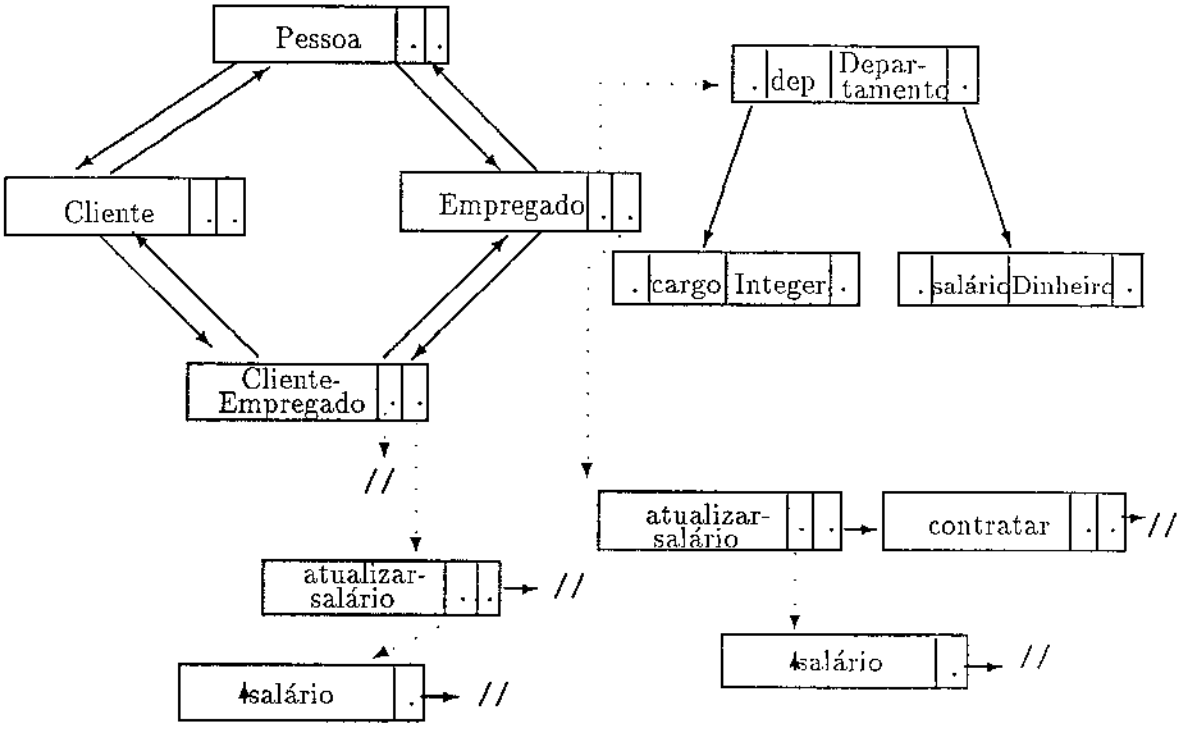


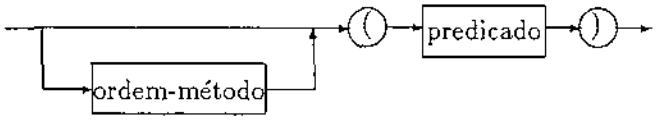
Figura 6.2: Estrutura de Dados Proposta

- estabelecimento de exceções às restrições (permanentes ou temporárias);
- estudo de efeito do diferimento de execução de regras na consistência do banco de dados, como, por exemplo, no sistema de [MD89], onde a execução de uma regra pode ser adiada.
- extensão da linguagem de programação do banco de dados para englobar definição de restrições, permitindo, assim, que o compilador faça a verificação preliminar da integridade. Esta solução é sugerida em [BLR91].

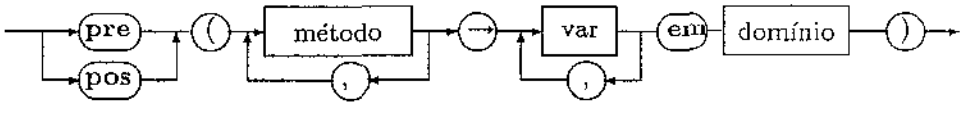
Apêndice A

A.1 Diagramas Sintáticos da Linguagem de Restrição

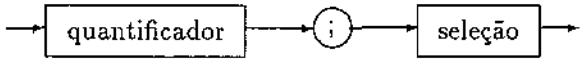
restrição:



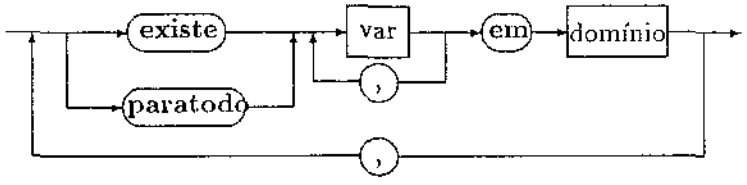
ordem-método:



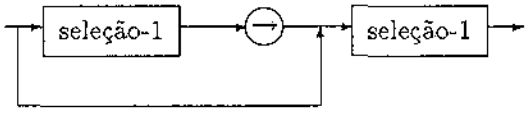
predicado:



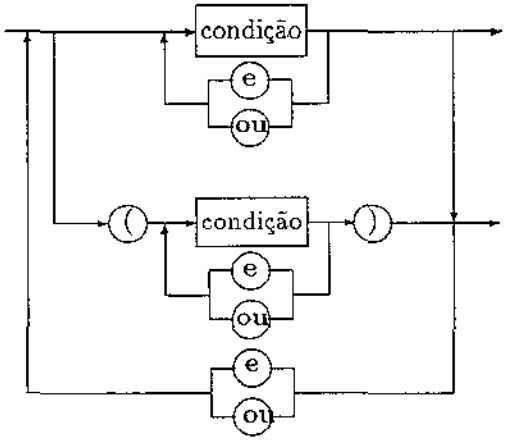
quantificador:



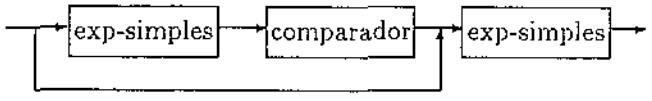
seleção:



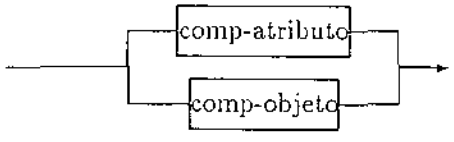
seleção-1:



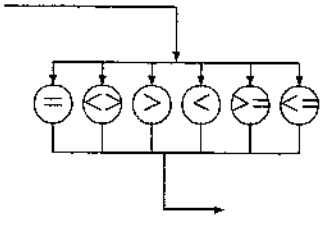
expressão:



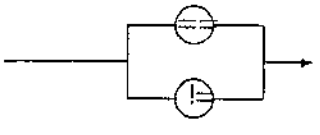
comparador:



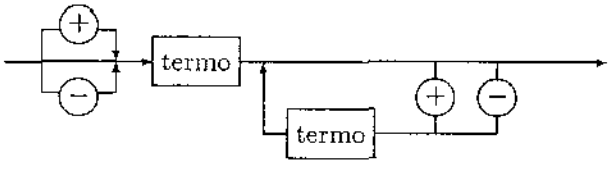
comp-atributo:



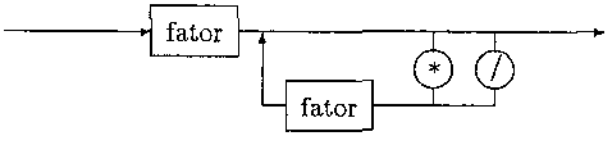
comp-objeto:



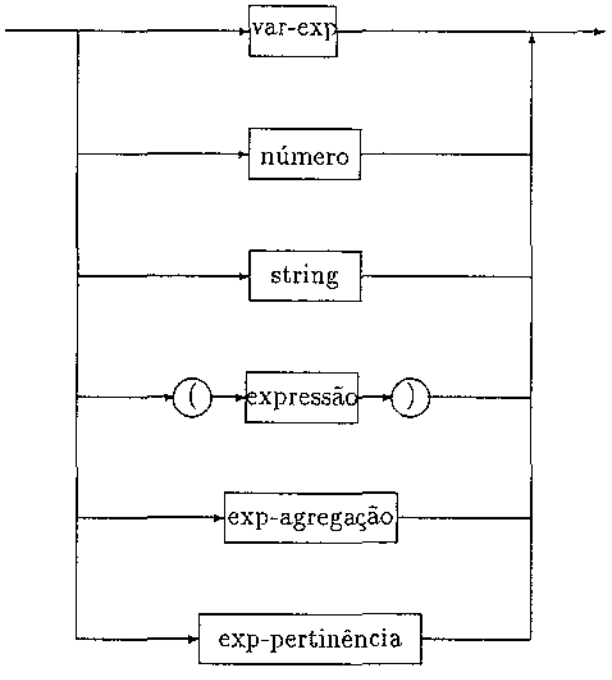
exp-simples:



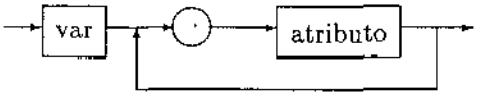
termo:



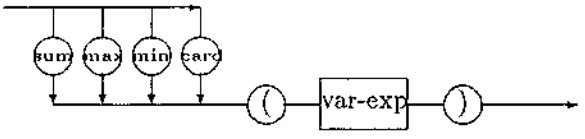
fator:



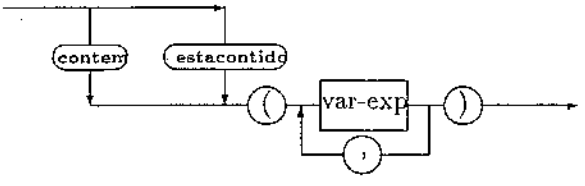
var-exp:



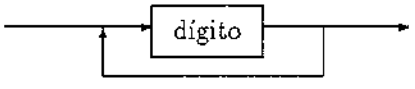
exp-agregação:



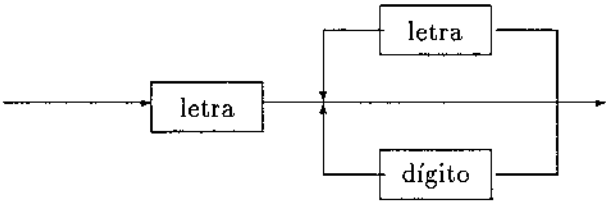
exp-pertinência:



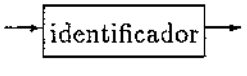
número:



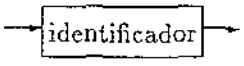
identificador:



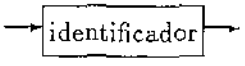
var:



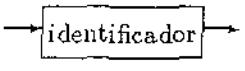
domínio:



atributo:



método:



Apêndice B

B.1 Algoritmos do sistema

Módulo: IDENTIFICA-VARIÁVEIS-ATRIBUTOS-CLASSES

Lê a restrição analisada sintaticamente pelo módulo ANÁLISE SINTÁTICA e identifica as variáveis, atributos e classes referenciados na restrição.

1. Ler a próxima palavra da restrição.
2. **Enquanto** não-fim restrição:
   (a) **Caso** palavra
      i. Variável: armazenar variável em uma lista simplesmente encadeada.
      ii. Atributo: gravar atributo em um arquivo (ARQUIVO-ATRIBUTO).
      iii. Classe: **para cada** variável da lista:
         A. Gravar a variável e a classe encontrada em um arquivo (ARQUIVO-VAR-CLASSE).
      iv. **pre** ou **pos**: gravar o nome do método e das classes referenciadas na condição em uma lista denominada LISTA-EVENTO. Ir para o fim do algoritmo.
   (b) Ler próxima palavra da restrição.

Módulo: CARREGA-INFORMAÇÕES-DO-ESQUEMA-DE-DADOS

Lê o esquema de dados e armazena as informações necessárias em estruturas de dados eficientes.

1. (Armazena as classes do esquema.)
   Inserir as classes do esquema em uma árvore AVL ordenada por nome de classe (ÁRVORE-CLASSE-ESQUEMA).

2. (Armazena os atributos e métodos das classes do esquema.)
   Para cada classe da ÁRVORE-CLASSE-ESQUEMA:

   (a) Inserir seus atributos e respectivos tipos a que pertencem em uma árvore AVL ordenada por nome de atributo (ÁRVORE-ATRIBUTO-CLASSE).

   (b) Inserir seus métodos em uma lista simplesmente encadeada ordenada por nome de método (LISTA-MÉTODO).

      i. (Armazena os atributos dos métodos.)
         Para cada método da lista:

         A. Inserir seus atributos e respectivos tipos a que pertencem em uma lista simplesmente encadeada ordenada por nome de atributo (LISTA-ATRIBUTO-MÉTODO).

   (c) (Armazena as superclasses das classes do grafo de herança.)
   Inserir as classes do grafo de herança em uma árvore AVL ordenada por nome de classe.

      i. **Para cada** classe da árvore:

         A. Inserir suas superclasses em uma lista simplesmente encadeada ordenada por nome de superclasse (LISTA-SUPERCLASSE).

   (d) (Armazena as subclasses das classes do grafo de herança.)
   Inserir as classes do grafo de herança em uma árvore AVL ordenada por nome de classe.

      i. **Para cada** classe da árvore:

         A. Inserir suas subclasses em uma lista simplesmente encadeada ordenada por nome de subclasse (LISTA-SUBCLASSE).

Módulo: GERA-EVENTOS

Gera os eventos responsáveis pela verificação da restrição, a partir de informações sobre o esquema de dados e sobre a restrição.

1. (Armazena as variáveis e suas respectivas classes em uma árvore de variáveis e as classes da restrição em uma árvore de classes.)
   Ler o ARQUIVO-VAR-CLASSE.

   (a) Inserir as variáveis e suas classes em uma árvore AVL ordenada por nome de variável (ÁRVORE-VAR-CLASSE).

   (b) Inserir as classes em uma árvore ordenada por nome de classe (ÁRVORE-CLASSE-RESTRIÇÃO).

2. (Pesquisa a classe dos atributos e de suas características extraídas e as armazena juntamente com os atributos na árvore de classes dos atributos da restrição, e na árvore de classes da restrição é armazenada apenas as classes encontradas.)
   Ler o ARQUIVO-ATRIBUTO.

   (a) Pesquisar a classe da variável (CVAR) que compõe o atributo na ÁRVORE-VAR-CLASSE.

   (b) Atribuir a CPESQ, CVAR.

   (c) **Para cada** característica extraída do atributo:

      i. Atribuir ID-ATRIBUTO, próximo identificador do atributo.

      ii. Chamar procedimento PESQUISA-CLASSE-DA-CARACTERÍSTICA-EXTRAÍDA (CPESQ, ID-ATRIBUTO).

      iii. Inserir a o atributo juntamente com a classe encontrada em uma árvore AVL ordenada por nome de atributo (ÁRVORE-CLASSE-ATRIBUTO-RESTRIÇÃO) e apenas a classe na ÁRVORE-CLASSE-RESTRIÇÃO.

3. (Verifica métodos das classes.)
   Para cada classe **c** da ÁRVORE-CLASSE-RESTRIÇÃO:

   (a) Pesquisar **c** na ÁRVORE-CLASSE-ESQUEMA.

   (b) Chamar procedimento VERIFICA-ATRIBUTOS-MÉTODO (**c**, **c**).

4. (Verifica métodos das superclasses.)
   Para cada superclasse **s** da LISTA-CLASSE-AFETADA:

   (a) Pesquisar **s** na ÁRVORE-CLASSE-ESQUEMA.

   (b) Chamar procedimento VERIFICA-ATRIBUTOS-MÉTODO (**s**, **c**).

5. (Propaga eventos para as subclasses.)
   Para cada classe **c** e método **m** da LISTA-EVENTO:

   (a) **Para cada** subclasse **s** da LISTA-SUBCLASSE da classe **c**:

      i. Inserir a subclasse **s** e o método **m** na LISTA-EVENTO.

   (b) Atribuir vazio à ÁRVORE-CLASSE-RESTRIÇÃO.

   (c) Armazenar as subclasses de **c** na ÁRVORE-CLASSE-RESTRIÇÃO.

   (d) **Repetir** este módulo a partir do passo 3 para a possível geração de eventos pelas subclasses inseridas na LISTA-EVENTO.

Procedimento: VERIFICA-ATRIBUTOS-MÉTODO (CPESQ, CGRAV)

Esse procedimento possui dois parâmetros: a classe de pesquisa e a classe de gravação. Nele é verificado se os métodos da classe CPESQ possuem pelo menos um atributo que está contido na ÁRVORE-CLASSE-ATRIBUTO-RESTRIÇÃO. Se o método possui, então gera o evento gravando a classe CGRAV e o método na LISTA-EVENTO.

1. **Para cada** método **m** da LISTA-MÉTODO da classe CPESQ:

   (a) **Se** algum atributo do método e sua respectiva classe **c** pertencem à ÁRVORE-CLASSE-ATRIBUTO-RESTRIÇÃO, **então** inserir a classe **c** e o método **m** em uma lista simplesmente encadeada ordenada por nome de classe (LISTA-EVENTO).

Procedimento: PESQUISA-CLASSE-DA-CARACTERÍSTICA-EXTRAÍDA (CPESQ, ID-ATRIBUTO)

Procura a classe a que pertence o parâmetro ID-ATRIBUTO, utilizando CPESQ como índice de pesquisa.

1. (Procura a classe CPESQ na árvore de classes do esquema.)
   Pesquisar CPESQ na ÁRVORE-CLASSE-ESQUEMA.

2. (Procura a classe do parâmetro ID-ATRIBUTO na árvore de atributos da classe.)
   Pesquisar na ÁRVORE-ATRIBUTO-CLASSE da classe CPESQ a classe de ID-ATRIBUTO.

3. **Se** encontrar a classe correspondente à característica extraída,
   então atribuir a CPESQ à classe encontrada.
   senão

   (a) Atribuir à classe CBASE, CPESQ.

   (b) **Enquanto** (não-fim LISTA-SUPERCLASSE da classe CPESQ ou não encontrar a classe da característica extraída):

      i. Atribuir a classe CPESQ a superclasse da LISTA-SUPERCLASSE.

ii. Chamar procedimento PESQUISA-CLASSE-CARACTERÍSTICA-EXTRAÍDA (CPESQ, ID-ATRIBUTO).

(c) (Registra a superclasse que contém a característica extraída como atributo.)
Se encontrar, **então** inserir a classe CBASE, a superclasse CPESQ, o atributo ID-ATRIBUTO e sua classe em uma lista simplesmente encadeada ordenada por nome de classe (LISTA-CLASSE-AFETADA).

Bibliografia

[ABCM83] M.P. Atkinson, P.J. Bailey, K.J. Chisholm P.W. Cockshott, and R. Morrison. An Approach to Persistent Proggraming. *The Computer Journal*, 26(4), 1983.

[ABD+89] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The Object-Oriented Database System Manifesto. In *Proceedings of the 1st International Conference on Deductive and Object-Oriented Databases*, Kyoto, Japan, December 1989.

[ACO85] A. Albano, L. Cardeli, and R. Orsini. GALILEO: A Strongly Typed, Interactive Conceptual Language. *ACM Transactions on Database Systems*, 10(2):230–260, 1985.

[Alt89] Altaïr, France. *The O_2 Programmer's Manual*, October 1989.

[BLR91] V. Benzaken, C. Lécluse, and P. Richard. Enforcing Integrity Constraints in Database Programming Languages. Technical Report 68-91, GIP Altaïr, INRIA, BP105, 78153 Le Chesnay, France, February 1991.

[Bor85] A. Borgida. Language Features For Flexible Handling of Exceptions in Information Systems. *ACM TODS*, 10(4):565–603, 1985.

[CFT88] M A. Casanova, A. L. Furtado, and L. Tucherman. A Monitor Enforcing Referential Integrity. *Simpósio Brasileiro de Banco de Dados*, (III):1–16, March 1988.

[Che76] P. Chen. An Entity–Relationship Model — Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1), January 1976.

[CN90] S. Chakravarthy and S. Nesson. Making an Object-Oriented DBMS Active: Design, Implementation, and Evaluation of a Prototype. In *Advances in Database Technology - EDBT International Conference on Extending Database Technology*, pages 393–406, Venice, Italy, March 1990.

[Coh89] D. Cohen. Compiling Complex Database Transition Triggers. In *Proceedings of the ACM SIGMOD, International Conference on Management of Data*, pages 225–234, May 1989.

[CW90] S. Ceri and J. Widom. Deriving Production Rules for Constraint Maintenance. In *Proc. 16th International Conference on Very Large Data Bases*, pages 566–577, Brisbane, Austrália, 1990.

[CW91] S. Ceri and J. Widom. Deriving Production Rules for Incremental View Maintenance. Technical report, IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120, March 1991.

[Dat86] C. J. Date. *Introdução a Sistemas de Bancos de Dados*. Campus, Brasil, 4.a edition, 1986.

[Dat89] C. J. Date. *An Introduction to Database Systems - Volume 2*. Addison-Wesley, USA, 1989.

[DBB+88] U. Dayal, B. Blaustein, A. Buchmann, U. Chakravarthy, M. Hsu, R. Ledin, D. McCarthy, A. Rosenthal, S. Sarin, M. J. Carey, M. Livny, and R. Jauhari. The HIPAC Project: Combining Active Database and Timing Constraints. *SIGMOD Record*, 17(1):51–70, March 1988.

[DBM88] U. Dayal, A. Buchmann, and D. McCarthy. Rule Are Objects Too: A knowledge Model For An Active, Object-Oriented Database System. In *Proc. 2nd workshop OODBS LNCS*, pages 129–143, West Germany, September 1988.

[Deu90] O. Deux. The Story of O_2. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):91–108, March 1990.

[DHL90] U. Dayal, M. Hsu, and R. Ladin. Organizing Long-Running Activities with Triggers and Transactions. In *Proc. ACM SIGMOD International Conf. on Management of Data*, May 1990.

[HK87] R. Hull and R. King. Semantic Database Modeling: Survey, Applications, and Research Issues. *ACM Computing Surveys*, 3(19):201–260, September 1987.

[HM81] M. Hammer and D. Mcleod. Database Description with a Semantic Data Model: SDM. *ACM Transactions on Database Systems*, 6(3), 1981.

[JMSS90] M. Jarke, S. Mazumdar, E. Simon, and D. Stemple. Assuring Database Integrity. *J. Database Adm.*, 1(1):391–400, 1990.

[KC86] S. Khoshafian and G.P. Copeland. Object Identity. In *ACM Proceedings of the Conference on Object-Oriented Programming Systems, Language and Applications*, Portland, OR, September 1986.

[KDM88] A. Kotz, K. Dittrich, and J. Mulle. Supporting Semantic Rules by a Generalized Event/trigger Mechanism. In *Proc. 1st EDBT*, pages 76–91, 1988.

[Kun85] C. Kung. On Verification of Database Temporal Constraints. In *Proceedings of the ACM SIGMOD, International Conference on the Management of Data*, pages 169–179, 1985.

[Laf82] G. M. E. Lafue. Semantic Integrity Dependencies and Delayed Integrity Checking. In *Proc. 8th International Conference on Very Large Data Bases*, pages 292–297, Mexico City, 1982.

[LSAS77] B. Liskov, A. Snyder, R. Atcinson, and C. Shaffert. Abstraction Mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, 1977.

[Md86] C. B. Medeiros and L. L. dÓliveira. Preprocessamento de Atualizações em Banco de Dados Relacionais. *IV Simpósio Brasileiro de Banco de Dados*, pages 253–263, 1986.

[MD89] D. R. McCarthy and U. Dayal. The Architecture of an Active Data Base Management System. In *Proceedings of the ACM SIGMOD, International Conference on the Management of Data*, pages 215–224, Portland, Oregon, jun 1989.

[Mor83] M. Morgenstern. Active Database as a Paradigm for Enhanced Computing Environment. In *Proc. Ninth International Conference in Very Large Database*, pages 34–42, Florence, Italy, October 1983.

[Mor84] M. Morgenstern. CONSTRAINT EQUATIONS: Declarative Expression of Constraints With Automatic Enforcement. In *Proc. Tenth International Conference in Very Large Database*, pages 291–300, Singapore, August 1984.

[MP91a] C. Medeiros and P. Pfeffer. Object Integrity Using Rules. In *Proceedings European Conference on Object-Oriented Programming*, pages 219–230, 1991.

[MP91b] C. B. Medeiros and P. Pfeffer. Transformação de um Banco de Dados Orientado a Objetos em um BD Ativo. *VI Simpósio de Banco de Dados*, pages 238–252, March 1991.

[NQZ90] R. Nassif, Y. Qiu, and J. Zhu. Extending The Object–Oriented Paradigm To Support Relationships and Constraints. Technical report, U S WEST Advanced Technologies, 6200 S. Quebec St. Suite 420 Englewood, CO 80122, USA, 1990.

[Shi81] D. Shipman. The Functional Data Model and the Data Language DAPLEX. *ACM TODS*, 1(6):140–173, March 1981.

[SHP89] M. Stonebraker, M. Hearst, and S. Potamianos. A Commentary on the Postgres Rules System. *SIGMOD Record*, 18(3):5–11, September 1989.

[SJGP90] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On Rules, Procedures, Caching and Views in Data Base Systems. In *Proceedings of the ACM SIGMOD, International Conference on Management of Data*, pages 281–290, Atlantic City, NJ, May 1990.

[SkS86] S. K. Sarin, C. W. kaufman, and J. E. Somers. Using History Information to Process Delayed Database Updates. In *Proc. 12th International Conference on Very Large Data Bases*, pages 71–78, Kyoto, 1986.

[Sny86] A. Snyder. Encapsularion and Inheritance in Object-Oriented Programming Languages. In *Proceedings of the Conference on Object-Oriented Programming Systems, Language and Applications*, Portland, OR, September 1986.

[SR88] T. Sellis and L. Raschid. Implementing Large Production Systems in a DBMS Environment: Concepts and Algorithms. In *Proceedings of the ACM SIGMOD, International Conference on Management of Data*, pages 281–290, Chicago, Illinois, June 1988.

[SRH88] M. Stonebraker, L. A. Rowe, and M. Hirohama. The POSTGRES Rule Manager. *IEEE Transactions on Knowledge and Data Engineering*, 14(7):897–907, July 1988.

[SRH90] M. Stonebraker, L. A. Rowe, and M. Hirohama. The Implementation of POSTGRES. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):125–142, March 1990.

[UD90] S. D. Urban and L. M. L. Delcambre. Constraint Analysis: A Design Process for Specifying Operations on Objects. *IEEE Transactions on Knowledge and Data Engineering*, 2(4):391–400, December 1990.

[WF90] J. Widom and S. J. Finkelstein. Set-oriented Production Rules in Relational Database Systems. In *Proceedings of the ACM SIGMOD, International Conference on the Management of Data*, pages 259–269, Atlantic City, NJ, May 1990.

[ZM90] S. B. Zdonik and D. Maier. *Readings in Object Oriented Database Systems.* Morgan Kaufmann, California, 1990.