UNIVERSIDADE FEDERAL DA PARAIBA

CENTRO DE CIENCIAS E TECNOLOGIA

COORDENAÇAO DE POS-GRADUAÇAO EM INFORMATICA

MARCOS BASTOS DAVID

DESCRIÇAO FORMAL DA ESTRUTURA DO MODELO ORIENTADO A OBJETOS TEMPORAL - TOM (Temporal Object Model)

Dissertação apresentada ao Curso de MESTRADO EM INFORMATICA da Universidade Federal da Paraiba, em cumprimento às exigências para obtenção do Grau de Mestre.

ULRICH SCHIEL

Orientador

D249d David, Marcos Bastos
Descricao formal da estrutura do modelo orientado a objetos temporal - TOM (Temporal Object Model) / Marcos Bastos David. - Campina Grande, 1992.
120 f.

Dissertacao (Mestrado em Informatica) - Universidade Federal da Paraiba, Centro de Ciencias e Tecnologia.

1. Modelo Orientado a Objetos Temporal 2. Banco de Dados 3. Base de Dados 4. Dissertacao - Informatica I. Schiel, Ulrich II. Universidade Federal da Paraiba - Campina Grande (PB) III. Título

CDU 004.658(043)

DESCRIÇÃO FORMAL DA ESTRUTURA DO MODELO ORIENTADO A OBJETOS TEMPORAL - TOM (Temporal Object Model)

MARCOS BASTOS DAVID

DISSERTAÇÃO APROVADA EM 19.02.1992

ULRICH SCHIEL - Dr.
Orientador

DECIO FONSECA - Dr.
Componente da Banca

JOSELUCE DE FARIAS CUNHA - M.Sc.
Componente da Banca

Campina Grande, 19 de fevereiro de 1992

Aos meus pais Waldir e Lourdes e aos meus irmãos. Pelo amor que lhes tenho e pelos meses que deixei de estar com eles para me dedicar a este trabalho.

AGRADECIMENTOS

- A DEUS que me deu inteligência e perseverança para que pudesse atingir meu objetivo;

- Ao meu orientador Dr. Ulrich Schiel, pela orientação, pelo encorajamento nas horas difíceis e, acima de tudo, pelos conhecimentos que me transmitiu, contribuindo significativamente para o meu desenvolvimento acadêmico;

- Aos demais colegas do DSC que me apoiaram e contribuiram direta ou indiretamente para a realização deste trabalho.

DESCRIÇÃO FORMAL DA ESTRUTURA DO MODELO ORIENTADO A OBJETOS TEMPORAL - TOM (Temporal Object Model)

RESUMO

O aumento da complexidade nas novas aplicações de bancos de dados (CAD/CAM, sistemas de informação de escritório multimídia, sistemas baseados em conhecimento, CASE, etc) exige um cuidado maior para manter a integridade das estruturas de dados complexas a serem manipuladas. O paradigma de orientação a objetos vem suprir os conceitos necessários à abordagem dessas aplicações avançadas, de modo mais modularizado e seguro.

O modelo proposto TOM (Temporal Object Model) é uma extensão do modelo semântico THM (Temporal-Hierarchic Data Model) com características de orientação a objetos. O modelo TOM é descrito por meio de um sistema de metaclasses, visando a um ambiente orientado a objetos aberto. Conceitos de tempo e versionamento de objetos também são características do modelo.

SUMARIO

1. Introdução

1.1. Descrição do Problema

Os sistemas de informação estão se tornando mais complexos a cada dia. A estrutura das novas entidades do mundo real está cada vez mais complexa, com atributos extremamente sub-estruturados. Os modelos clássicos (hierárquico, de rede e relacional) se mostram limitados para representar tais entidades. Considere, por exemplo, a tarefa de representar as partes componentes de um robô usando o modelo relacional. De um lado, temos a simplicidade do modelo relacional, com seus conceitos de relação, tupla e domínio atômico. Do outro lado aparece a intrincada estrutura de um robô, com seus milhares de componentes eletrônicos, sensores, engrenagens interligadas entre si e uma infinidade de sub-componentes de cada uma dessas peças. E fácil perceber que a representação da entidade ROBO por meio de relações será algo no mínimo difícil de entender. E mais: as operações de recuperação e atualização sobre esse esquema de relações exigirão um conhecimento sobre-humano da estrutura do robô.

E por que não usar um modelo semântico? E certo que os modelos de dados semânticos permitem representar uma entidade complexa através de um simples objeto, no banco de dados. Os modelos semânticos geralmente já incluem conceitos de relacionamentos hierárquicos entre classes de entidades, como a generalização e a agregação. Agora a representação da nossa entidade ROBO pode ser mais fácil de entender e manipular.

Contudo, os modelos semânticos de dados carecem de certas características inerentes aos modelos de dados orientados a objetos. Entre os principais princípios que estão ausentes nos modelos semânticos, destacam-se:

A) Objetos Complexos

Segundo [DITT87], existem três níveis de orientação a objetos. Estes níveis representam os diversos graus em que um sistema de banco de dados pode ser considerado orientado a objetos. Os três níveis de orientação a objetos, em ordem crescente, são:

. O Nível Estrutural - caracterizado pela presença de objetos complexos, geralmente definidos a partir da estrutura de tipos de objetos primitivos pré-definidos ou definidos pelo usuário;

. O Nível Operacional - onde pode-se perceber a associação da estrutura de objetos simples (tipos de dados abstratos) com as operações permitidas sobre esses objetos. Assume-se que um tipo de dado abstrato pode ser manipulado somente pelas operações definidas sobre aquele tipo;

. O Nível de Comportamento - no qual podem ser definidos novos tipos de objetos complexos (nível estrutural) juntamente com as operações permitidas sobre os mesmos (nível operacional).

Os modelos de dados semânticos podem ser classificados entre os modelos estruturalmente orientados a objetos. As estruturas dos objetos complexos modelam as entidades complexas

do mundo real, porém estas estruturas de dados são completamente independentes das operações que as manipulam. Consequentemente, os objetos complexos num banco de dados semântico são relativamente desprotegidos e vulneráveis à execução de operações não autorizadas ou não adequadas à sua estrutura.

B) Modularidade, Extensibilidade e Reusabilidade

Como foi visto anteriormente, os modelos semânticos apresentam-se pouco modulares. O princípio de encapsulamento é ignorado nos modelos de dados semânticos. As operações são independentes das classes de objetos, i.e., são definidas para qualquer tipo de objeto. Desta forma, a camada de proteção sobre a estrutura de objetos complexos é visivelmente fraca em modelos semânticos.

O princípio de **extensibilidade** torna possível a definição de novos tipos de objetos a partir dos tipos de objetos pré-definidos, e a utilização dos tipos definidos pelo usuário indistintamente dos tipos primitivos, na formação de novos tipos de objetos. Considera-se que a definição de um novo tipo de objeto inclui a definição das respectivas operações.

O conceito de extensibilidade de tipos de objetos permite realizar a extensibilidade de software. Novos requisitos podem ser atendidos facilmente através da extensão do sistema de software a partir dos módulos (objetos) primitivos. A estratégia de utilização de módulos previamente definidos para estender o sistema é denominada **reusabilidade.**

Os modelos de dados semânticos apresentam

extensibilidade de tipos mas não de operações. Por isso, a implementação de novos requisitos de procedimento é muitas vezes proibitiva.

C) Segurança

Com o aumento da complexidade da estrutura dos objetos, surge a necessidade de definir cuidadosamente as operações sobre estes objetos, a fim de assegurar a integridade do banco de dados e a consistência dos dados. Pelo princípio do encapsulamento, os modelos orientados a objetos permitem armazenar a estrutura dos objetos complexos, junto com as operações permitidas sobre os mesmos, dentro do banco de dados. Os programas de aplicação precisam apenas chamar as operações previamente definidas, com a certeza de que a consistência e a integridade dos objetos serão preservadas. Assim, podemos implementar um nível maior de segurança através da estrutura de objetos complexos.

Os modelos semânticos, por não apresentarem encapsulamento de dados e operações, são desprovidos deste nível de segurança.

1.2. Histórico dos Modelos de Dados

Na aurora da ciência da computação não havia muita preocupação com a semântica das aplicações. O pessoal de processamento de dados lidava basicamente com uma coleção de dados organizados como arquivos manuais: eram os **modelos primitivos.** A geração seguinte começou a dar importância a tipos mais sofisticados de consulta ao banco de dados e tentou descobrir novas formas de armazenar os dados de maneira a

facilitar tais consultas. Surgiram os **modelos clássicos.** Ainda preocupados em aumentar o poder de representação do mundo real dentro de um banco de dados, os pesquisadores idealizaram os **modelos semânticos.** Nos últimos anos, novos conceitos têm sido descobertos e uma nova filosofia foi aplicada aos modelos semânticos, dando origem aos **modelos orientados a objetos.**

1.2.1. Modelos de Dados Primitivos

Nos modelos de dados primitivos, os dados da organização são armazenados em um grande banco de dados. Este banco de dados, por sua vez, é uma coleção de arquivos interrelacionados por ponteiros. Os arquivos são compostos de registros, e cada registro é formado de campos. Cada campo tem um certo tipo e um certo tamanho. Os programas de aplicação podem apenas ler ou gravar registros.

1.2.2. Modelos de Dados Clássicos

O primeiro modelo clássico a surgir foi o **modelo hierárquico,** como uma extensão dos modelos primitivos. Ele usa a nomenclatura de **segmentos** e **campos** para denotar registros e itens de dado, respectivamente. Um BD hierárquico pode ser visto como um conjunto de árvores de segmentos. A característica principal do modelo hierárquico é que uma ocorrência de segmento pai pode ter várias ocorrências de segmento filho, mas uma ocorrência de segmento filho pode ter apenas uma ocorrência de segmento pai. Não pode haver uma ocorrência de segmento filho sem um pai, daí a denominação de modelo hierárquico. A principal limitação do

modelo hierárquico é que ele foi projetado para representar apenas relacionamentos 1:n. Com isso, ele se torna extremamente desagradável quando tentamos representar relacionamentos n:n.

Para sanar as deficiências do modelo hierárquico, surgiu o **modelo de rede**, considerado o segundo modelo clássico. Este modelo está fundamentado em dois conceitos: o tipo registro (RECORD TYPE) e o tipo conjunto (SET TYPE). Um tipo de conjunto define uma lista encadeada de registros de um mesmo tipo. E possível a representação de relacionamentos 1:1, 1:n, n:n, bastando definir os conjuntos de registros apropriados entre os tipos de registro apropriados. O modelo de rede é assim denominado devido às listas de registros encadeados que representam internamente os tipos de conjuntos.

O modelo clássico mais recente é o **modelo relacional**. O principal conceito deste modelo é a **relação**, derivada da definição matemática de uma relação entre domínios D1, ..., Dn como um subconjunto do produto cartesiano D1 x ... x Dn. No modelo relacional, uma relação é um conjunto de n-tuplas, onde cada tupla representa um relacionamento n-ário.

Devido à sua simplicidade, o modelo relacional facilita muito a manipulação e o acesso ao banco de dados. Entretanto, para o projetista do BD a representação de toda a semântica das aplicações apenas através de tabelas é algo um tanto complexo.

1.2.3. Modelos de Dados Semânticos

O principal objetivo dos modelos de dados é diminuir a diferença semântica existente entre o mundo real e o mundo do Banco de Dados (BD), permitindo representar ao máximo a semântica

da realidade dentro do próprio BD.

No intuito de melhorar a captação da semântica das aplicações, surgiram os primeiros **modelos semânticos de dados.** Entre os modelos semânticos existentes, destacam-se: o modelo de entidades e relacionamentos (E-R) [CHEN76], os modelos de relacionamentos binários [BN78], o modelo relacional estendido (RM/T) [CODD79], o modelo de dados semântico (SDM) [HM81], o modelo de dados funcional [SHIP81], o modelo de dados de eventos [KM82], o modelo hierárquico-temporal (THM) [SCHI82A].

O modelo relacional estendido (RM/T), por exemplo, apresenta uma série de refinamentos e melhorias ao modelo relacional de Codd. Entidades são representadas por relações-E (relações de entidades) e relações-P (relações de propriedades), sendo que as primeiras indicam a existência das entidades e as últimas representam certas propriedades das mesmas. Um tipo é representado como uma relação-E. Há três tipos de relacionamentos hierárquicos: associação (n:n), designação (n:1) e característica (dependência de existência). O modelo RM/T permite representar também a hierarquia de generalização através de hierarquias tipo/subtipo.

O modelo de eventos representa a realidade através de objetos e eventos de aplicação (transações), sendo que ambos são classificados como tipos de objeto. Este modelo suporta dois tipos de relacionamentos entre classes: generalização e agregação.

O modelo de entidades e relacionamentos representa o mundo real por meio dos conceitos de entidade, relacionamento e

atributo. Os modelos de relacionamentos binários utilizam apenas os conceitos de entidades e relacionamentos, evitando a distinção entre atributo e relacionamento.

1.2.4. Modelos Orientados a Objetos

A abordagem de orientação a objetos em bancos de dados surgiu das pesquisas sobre modelos semânticos de dados. Uma nova filosofia de implementação de sistemas de informação foi aplicada aos modelos semânticos, adicionando-lhes as características de encapsulamento e passagem de mensagens.

As características gerais de um modelo orientado a objetos são: objetos complexos, identidade de objetos, encapsulamento, tipos ou classes, herança, "overloading", amarração dinâmica, extensibilidade e completude computacional [BDZ89].

Estas características serão explicadas em maiores detalhes no capítulo 2.

Atualmente, há uma forte tendência no sentido dos chamados modelos abertos orientados a objetos. A definição de um modelo aberto será dada no capítulo 5.

O TOM (Temporal Object Model) é um exemplo de modelo orientado a objetos com tempo e versionamento. O termo **versionamento** será usado para denominar o modelo de tratamento das versões do processo de desenvolvimento de um objeto.

1.3. Objetivos Específicos da Pesquisa

O objetivo geral dos modelos de dados orientados a objetos é permitir a modelagem de sistemas mais modulares, através do encapsulamento de dados e métodos em tipos de dados abstratos (objetos).

Com o aumento da complexidade da estrutura dos objetos, há necessidade de definir cuidadosamente as operações sobre estes objetos, com a finalidade de preservar a consistência dos dados e a integridade do esquema conceitual. Essas operações bem definidas são incluídas juntamente com as estruturas de dados dos objetos dentro do banco de dados.

O encapsulamento da estrutura dos objetos com as operações fornece maior proteção e segurança sobre a estrutura dos objetos complexos. Estes objetos "protegidos", por sua vez, podem ser usados como primitivas na modelagem de sistemas mais extensíveis e receptíveis a alterações de requisitos.

Os modelos de dados em geral apresentam um conjunto rígido de conceitos e abstrações hierárquicas que são usados para modelar as aplicações de banco de dados. Se, num dado instante, a modelagem do problema exigir um conceito não previsto no modelo, então teremos uma limitação na representação da semântica do mundo real. Os modelos de dados que se constituem num conjunto fixo de conceitos são denominados **modelos fechados** (e.g., modelos clássicos tais como o modelo de rede e o relacional).

Recentemente surgiu a idéia de desenvolver **modelos de dados abertos**, de acordo com os princípios do segundo Modelo de Referência para SGBDs ANSI/SPARC [ANSI86]. Tais modelos permitem

extensões sobre seus conceitos e a criação de novas abstrações necessárias para aplicações de propósito específico. Em consequência disso, os modelos de dados abertos são capazes de se adaptar a qualquer aplicação de propósito geral ou especial.

Os objetivos específicos da pesquisa são:

[a] Descrever um metanível que permite a definição de modelos de dados;

[b] Fornecer meios de encapsulamento de dados e métodos (classes + operações do modelo THM(Temporal Hierarchic Data Model), criando o modelo TOM (Temporal Object Model);

[c] Definir a interface dos objetos através de métodos primitivos do TOM (Temporal Object Model);

[d] Estender os aspectos temporais do modelo THM para aspectos de versionamento de objetos;

[e] Definir a sintaxe e a semântica do modelo de passagem de mensagens do TOM;

[f] Descrever um sistema de metaclasses visando a um ambiente orientado a objetos aberto.

1.4. Esboço da Tese

O presente trabalho encontra-se organizado em capítulos, seções e subseções. No capítulo 2 serão apresentados e explicados os conceitos e características que identificam um modelo de dados orientado a objetos. No capítulo 3 será introduzida uma breve descrição do modelo TOM num nível apenas

informal, com o objetivo de dar ao leitor uma visão das características gerais inerentes ao modelo em questão. No capítulo 4, o modelo TOM será descrito de maneira mais rigorosa através de um formalismo axiomático. No capítulo 5, será introduzido o metanível e apresentado o meta-esquema do modelo TOM. No capítulo 6, será ilustrada a modelagem do modelo relacional, usando o metanível. O capítulo 7 apresentará as conclusões e sugestões de trabalhos futuros.

2. Conceitos Básicos de Orientação a Objetos

2.1. Introdução

Há muita polêmica sobre as características que deveriam ser essenciais a um sistema de banco de dados orientado a objetos. Vários pesquisadores já teceram suas considerações sobre o que seria um sistema orientado a objetos, mas sempre se estabelece uma margem para futuros questionamentos desses conceitos.

Nós descrevemos aquelas características que alcançaram um consenso entre os grandes especialistas no assunto, e.g., Bancilhon, Dittrich, Zdonik entre outros [BDZ89]. Segundo esses autores, um sistema de banco de dados orientado a objetos deveria ser um SGBD e um sistema orientado a objetos. Para ser um sistema orientado a objetos, o sistema de banco de dados deve incorporar as seguintes noções importantes: objetos, classes(ou tipos), encapsulamento, herança, mensagens, polimorfismo, amarração dinâmica, completude computacional e persistência de objetos.

Estes conceitos serão expostos em detalhes nas seções que se seguem. Na seção 2.2, será introduzida a noção de objeto e a sua relação com o conceito de classe. Na seção 2.3, serão dadas as definições de classe, encapsulamento e herança. A seção 2.4 descreverá a forma geral de uma mensagem, introduzindo os conceitos de polimorfismo, overloading e amarração dinâmica. E, finalmente, na seção 2.5 serão descritas as noções de completude computacional e persistência de objetos.

2.2. Objeto e Classe

A definição de objeto corresponde à noção de tipo de dado abstrato. Todo objeto é formado por uma parte de interface e uma parte de implementação. Na interface são definidas todas as operações que podem ser realizadas sobre aquele tipo de objeto. A parte de implementação é composta das estruturas de dados que constituem o objeto e das "procedures" que implementam as operações da interface. A estrutura de dados do objeto é construída a partir de tipos de objetos primitivos (inteiros, reais, booleanos, caracteres, strings, etc) usando construtores de objetos complexos, e.g., tupla, conjunto e lista(ou array). Estes construtores podem ser usados ortogonalmente para criar novos objetos complexos como, por exemplo, listas de conjuntos de tuplas, conjuntos de tuplas de listas, e assim por diante. Os novos tipos de objetos podem ser usados da mesma forma que os tipos primitivos na formação de novos tipos de objetos. As operações definidas na interface devem permitir a manipulação desses objetos complexos como um todo.

Todo objeto possui um identificador único, implícito, usado como chave primária(um ponteiro para o objeto). Esta afirmação implica que:

[1] O objeto tem uma existência independente do seu valor [BDZ89];

[2] Um mesmo objeto pode ser referenciado como componente de mais de um objeto;

[3] Atualizações sobre um objeto componente são refletidas para todos os objetos que referenciam aquele

componente;

[4] O programador não precisa se preocupar em gerar chaves únicas nem em manter a integridade referencial(algumas das desvantagens do modelo relacional).

As noções de classe e de tipo têm sido tratadas como sinônimas, na literatura sobre programação orientada a objetos. Contudo, existe uma diferença sutil entre esses dois conceitos.

A definição de tipo de objeto está relacionada à definição de tipos de dados abstratos. Um tipo de objeto é uma abstração representada por uma parte de interface e uma parte de implementação. A interface relaciona todas aquelas operações que podem ser efetivadas sobre objetos daquele tipo. A parte de implementação é formada por uma parte de estruturas de dados e uma parte contendo as procedures que implementam as operações da interface. As estruturas de dados compõem a estrutura de um objeto complexo do tipo em questão. A noção de tipo implica numa abstração mais intencional que permite, por exemplo, a checagem de tipo em tempo de compilação.

O conceito de classe de objetos tem um significado mais extencional do que o conceito de tipo. Isto implica que classes são usadas em tempo de execução para gerar novas instâncias ou armazenar instâncias de objetos daquela classe. Embora a estrutura básica de uma classe seja a mesma de um tipo de objeto, a noção de classe é usada mais em tempo de execução para criar e manipular novas instâncias, tornando difícil a checagem de tipo em tempo de compilação.

Os modelos orientados a objetos atualmente existentes

incorporam uma das duas noções(tipo ou classe) ou ambas.

2.3. Classes, Encapsulamento e Herança

Uma classe de objetos é uma abstração de diversos objetos individuais com estrutura e comportamento semelhantes. A definição de classe(ou tipo) de objetos traz um conceito muito importante tanto em programação orientada a objetos como em bancos de dados orientados a objetos: a noção de **encapsulamento.** Como foi visto na seção anterior, uma classe de objetos é uma abstração composta de duas partes: uma parte de interface e uma parte de implementação. A única parte visível para o usuário da classe são os nomes das operações(métodos) da interface. Qualquer objeto que seja instância daquela classe só poderá ser manipulado por meio dos métodos definidos na interface. Isto fornece um bom nível de proteção para os objetos, assegurando a consistência e a integridade de suas estruturas complexas e proporcionando um dispositivo de autorização de acesso.

A noção de encapsulamento é particularmente revolucionária na área de bancos de dados. Nos SGBDs tradicionais e mesmo nos modelos semânticos, as descrições dos dados são guardadas dentro do próprio banco de dados, enquanto que os programas de aplicação (operações) são codificados a parte. A medida que as entidades do mundo real se tornam mais complexas, surge a necessidade de projetar mais cuidadosamente as operações sobre estes objetos, a fim de manter sua integridade. Tais operações(métodos) são então guardadas junto com a estrutura dos dados, no banco de dados. Os programas de aplicação, por sua vez,

apenas chamam os métodos previamente definidos sobre os objetos complexos. Qualquer alteração no comportamento dos métodos não afeta os programas de aplicação.

As classes de objetos são organizadas numa estrutura de árvore (ou grafo) hierárquica. Cada classe possui uma superclasse. Os dados e métodos definidos na superclasse são herdados pela subclasse. Por isso, o conjunto de dados e métodos de uma classe é a união do conjunto de dados e métodos definidos na própria classe mais os dados e métodos herdados de sua(s) superclasse(s). O mecanismo de herança proporciona reusabilidade do código e dos dados definidos nas superclasses, simplificando a codificação dos métodos e evitando a redundância nas descrições de dados das classes de objetos.

Quando se considera uma hierarquia de classes, dois conflitos podem ocorrer:

[1] Conflito entre uma classe e sua superclasse

Quando temos métodos com o mesmo nome na classe e na superclasse, é dada precedência para a classe;

[2] Conflito entre superclasses de uma mesma classe

Já vimos que uma classe herda os atributos e métodos de suas superclasses. Quando há um atributo ou método com o mesmo nome em superclasses diferentes, é preciso escolher de qual das superclasses será herdado o atributo/método. Esta escolha pode ser implementada de duas maneiras:

[a] Estabelecendo uma lista de precedência de superclasses ou;

[b] Deixando o usuário escolher a superclasse em

tempo de execução.

O tipo de conflito descrito no item [2] acima é denominado herança múltipla. O modo de resolução deste conflito, descrito no sub-item [a], pode gerar herança indesejável.

2.3.1. Metaclasses ou Categorias

Um outro conceito interessante em sistemas orientados a objetos é a noção de metaclasses. Ao contrário do que se poderia pensar, metaclasses não têm nada a ver com superclasses. O mecanismo de metaclasses possibilita a definição de categorias de classes pré-definidas a partir das quais novas metaclasses podem ser criadas, como instâncias das anteriores. Neste contexto, uma metaclasse(categoria) poderia ser vista como uma classe cujas instâncias são outras classes de objetos, e não instâncias de objetos diretamente. De fato, as metaclasses dispõem de um método de criação de instâncias, semelhante ao método existente para classes (new), que permite a criação de novas instâncias de classes (newClass). O sistema de hierarquia de metaclasses assegura a herança de atributos e métodos também entre classes e metaclasses. A figura 2.1. ilustra a hierarquia de metaclasses.

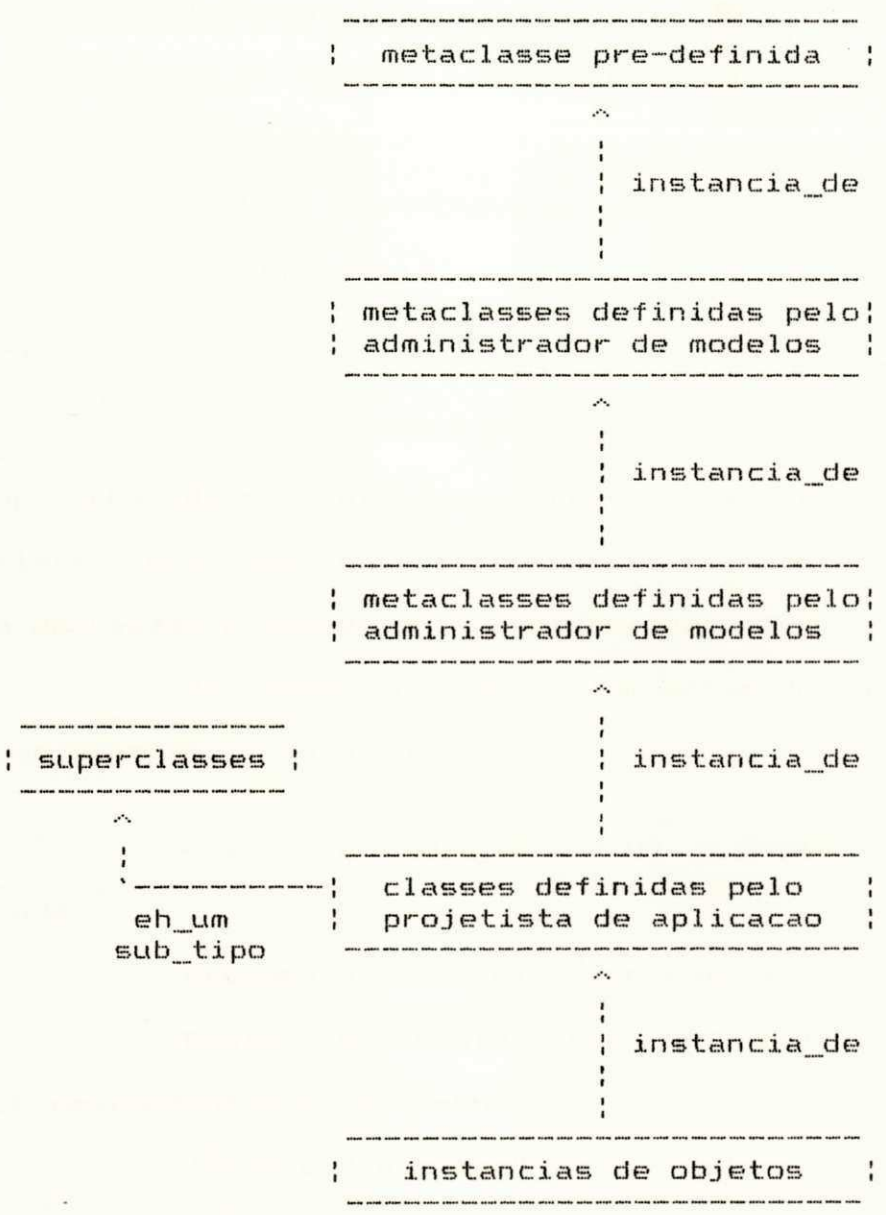


Figura 2.1 - Hierarquia de Metaclasses

2.4. Mensagens, Polimorfismo e Amarração Dinâmica

As mensagens em um sistema orientado a objetos desempenham o mesmo papel das chamadas a procedure, num ambiente convencional. O formato geral de uma mensagem envolve 3 componentes básicos:

1- O identificador do objeto receptor da mensagem;

2- O seletor da operação a ser realizada sobre o objeto;

3- Os argumentos a serem usados nesta operação (opcional).

Para efetivar uma operação sobre um dado objeto, é preciso enviar uma mensagem a este objeto, indicando qual das operações da sua interface será executada e, opcionalmente, uma lista de parâmetros a serem usados pela procedure que implementa a operação. O método então retorna um valor ao objeto chamador.

Mensagens podem ser compostas de mais de um seletor. Por exemplo, seja a operação:

Orçamento Gasto: 30,00 Conta: "refeição"

onde

Orçamento é o objeto receptor;

Gasto e **Conta** são os seletores dos métodos correspondentes em **Orçamento**;

30,00 e **"refeição"** são os valores dos parâmetros reais passados para os métodos **Gasto** e **Conta**, respectivamente.

Quando um objeto recebe uma mensagem, ocorre uma checagem do método selecionado sobre a classe do objeto receptor. Se a classe do objeto receptor não contiver o método especificado, a superclasse será checada em busca do método em questão, e assim sucessivamente. A execução de um método em um dado objeto geralmente envolve o envio de mensagens para outros objetos, a fim de executar outros métodos, gerando uma cadeia de

mensagens que só termina quando os objetos de tipos primitivos são referenciados.

As propriedades de polimorfismo, overloading e amarração dinâmica de um ambiente orientado a objetos podem ser melhor descritas por meio de um exemplo. Considere o seguinte problema: imprimir um conjunto de objetos de tipos diferentes. Num ambiente convencional nós teríamos de codificar uma rotina de impressão diferente para cada tipo de objeto e chamar a rotina apropriada de acordo com o tipo do objeto. Como consequência, o programador da aplicação precisa conhecer todos os tipos possíveis a priori, para que possa prever um formato de impressão adequado, numa declaração "case".

Um ambiente orientado a objetos possibilita o uso de um único identificador de operação de impressão para designar procedimentos diferentes, conforme o tipo do objeto corrente. Cada tipo(classe) de objeto tem um método de impressão diferente com o mesmo nome. Tal estratégia é denominada **polimorfismo.** A utilização de um único operador sobre objetos de tipos diferentes é chamada de **overloading** de operador. Durante a execução do programa, a mensagem de impressão é enviada para objetos diferentes e, de acordo com o tipo do objeto em questão, o corpo da procedure de impressão adequada é ligada ao nome do seletor, na mensagem. Esta ligação em tempo de execução é conhecida na literatura especializada com o nome de **amarração dinâmica.**

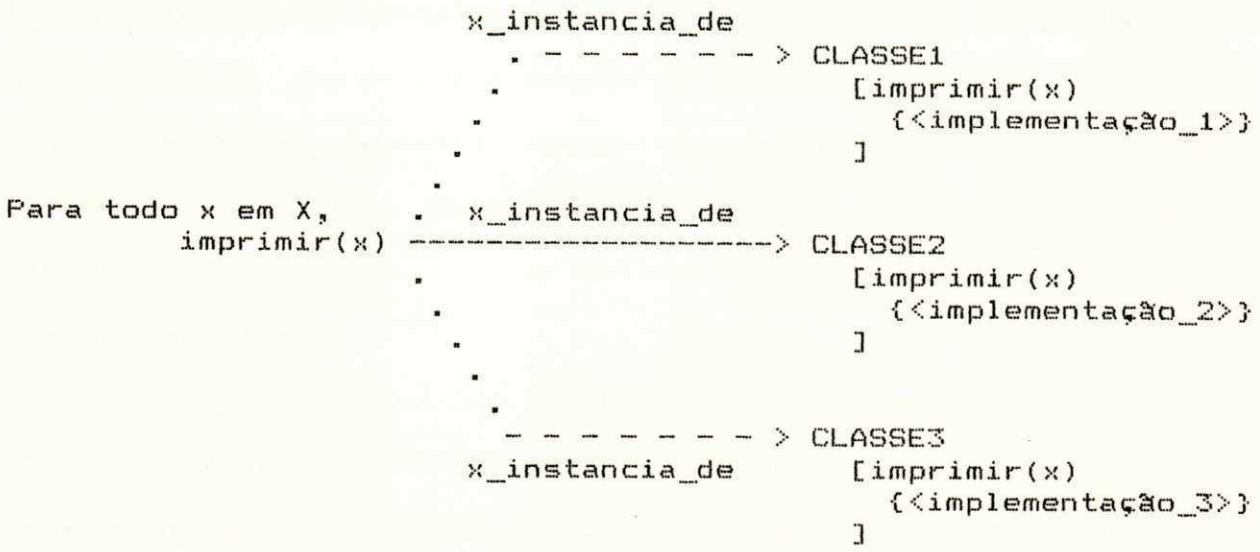


Figura 2.2 - Ilustração de polimorfismo, overloading e amarração dinâmica

O esquema acima descrito facilita a tarefa de manutenção de programas. Se um novo tipo de objeto tiver de ser incluído no conjunto, não será necessário alterar um "case" no programa: a própria definição do tipo já incluirá um método de impressão para o novo tipo de objeto.

2.5. Completude Computacional e Persistência de Objetos

A característica de completude computacional de uma linguagem assegura que ela possa realizar qualquer função computável. As linguagens de manipulação e consulta a bancos de dados em geral são computacionalmente incompletas. Estas linguagens específicas precisam ser usadas em conjunto com linguagens hospedeiras convencionais que forneçam maior flexibilidade computacional. Há uma forte tendência em sistemas de bancos de dados orientados a objetos em direção ao desenvolvimento de novas linguagens de consulta que sejam

computacionalmente completas, ou novas linguagens convenvionais que permitam consultas a bancos de dados. Uma das principais dificuldades encontradas nesse empreendimento é a diferença semântica existente entre as linguagens convencionais e as linguagens de consulta: as primeiras são procedurais e as últimas são declarativas.

Persistência de objetos é uma propriedade óbvia em sistemas de bancos de dados. Em linguagens de programação, os objetos usados no contexto de um módulo são perdidos no final do processo, i.e., não há persistência de objetos. Já em sistemas de bancos de dados, esses mesmos objetos são armazenados num banco de dados global, permitindo sua persistência após o término de uma transação e consequente reutilização em outros processos. O conceito de persistência de objetos foi herdado pelos novos sistemas de bancos de dados orientados a objetos.

3. O Modelo de Dados Orientado a Objetos TOM

3.1. Introdução

A estrutura do modelo de dados orientado a objetos temporal TOM(Temporal Object Model) herda muitas das características do modelo semântico THM (Temporal-Hierarchic Data Model)[SCHI83]. Por isso, o entendimento da descrição do TOM exige o conhecimento de alguns conceitos inerentes ao THM.

A base do modelo TOM parte do princípio de que o universo é dividido em três mundos: o mundo abstrato, o mundo concreto e o mundo do modelo. O universo de discurso é a parte do mundo real(concreto e abstrato) que é relevante para uma aplicação específica [SCHI82A]. O resultado do processo de modelagem é a representação do mundo real por meio das abstrações do modelo.

O modelo TOM encontra-se estruturado sobre três pilares principais: hierarquia, tempo e versionamento, e dinâmica. Os dois primeiros fatores determinam o aspecto estático do modelo e o terceiro item reflete seu aspecto dinâmico.

A próxima seção expõe o modelo em maiores detalhes. Na seção 3.2 será descrito o modelo TOM segundo seus aspectos estáticos e dinâmicos. Aqui nós nos limitamos a dar uma descrição informal do modelo. Mais tarde será introduzida uma descrição mais formalizada do TOM.

3.2. Visão Geral dos Conceitos do Modelo TOM

O modelo de dados orientado a objetos temporal (TOM - Temporal Object Model) é um modelo orientado a objetos baseado nos conceitos de classe, relacionamento e método. Toda e qualquer entidade do mundo real é representada por um objeto instância de uma classe, no mundo do modelo. Cada classe de objetos apresenta uma estrutura de relacionamentos e métodos que é herdada por todos os objetos daquela classe. Toda classe tem um nome e uma descrição de suas instâncias. Esta descrição é fornecida pela declaração de um conjunto de **relacionamentos instância** e um conjunto de **métodos instância.** São os seguintes os tipos de relacionamentos que podem estar presentes na definição de uma classe:

Relacionamento-instância ==> é aquele que varia de instância para instância. Por exemplo, o relacionamento tem_nome relaciona objetos da classe EMPREGADO com objetos da classe NOME. Toda instância da classe EMPREGADO possui um nome em NOME.

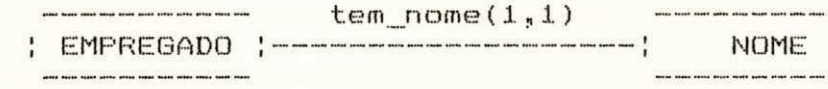


Figura 3.1 - Relacionamento-instância

Relacionamento-classe ==> é aquele que se relaciona à classe como um todo, i.e., não varia de instância para instância. Os relacionamentos-classe servem a duas finalidades:

1) Considerar a classe como um objeto com seus próprios relacionamentos;

2) Fatorar valores comuns a todas as instâncias.

Por exemplo, o relacionamento consumo-maximo relaciona a classe FABRICANTE a uma instância da classe CONSUMO_COMBUSTIVEL.

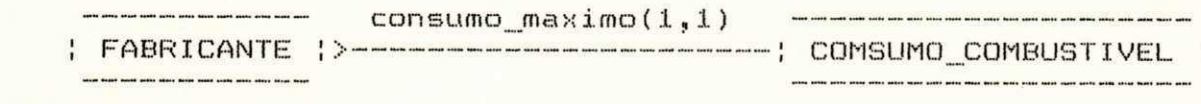


Figura 3.2 - Relacionamento-classe

A cada relacionamento é associada uma cardinalidade, expressa pelo par de valores (n,m), o qual significa que cada membro da primeira classe está associado a no mínimo n e no máximo m membros da segunda classe.

São os seguintes os tipos de métodos que podem aparecer na descrição de uma classe:

Método-instância --> é aquele que é usado numa mensagem enviada a uma instância de objeto.

Método-classe --> é aquele que é usado numa mensagem enviada a uma classe de objetos. Os métodos-classe são usados para manipular relacionamentos-classe, criar novas instâncias, ou selecionar instâncias de uma classe.

Uma classe é um sistema composto de duas partes:

1) A descrição da classe - composta do nome da classe, uma lista de relacionamentos, uma lista de métodos, informação sobre sua posição na hierarquia de classes e parâmetros de tempo;

2) O conjunto de instâncias - composto de um conjunto de objetos que herdam a lista de relacionamentos e métodos da classe a que pertencem.

Consideramos quatro tipos de classes de objetos, quanto à identificação:

1) Classes com objetos identificados por um relacionamento-chave. Por exemplo, as instâncias da classe FABRICANTE são identificadas pelo relacionamento **tem_nome** com a classe NOME_FABRICANTE;

2) Classes permitindo vários objetos idênticos. Instâncias são recuperadas selecionando-se algumas propriedades. Por exemplo, instâncias da classe PESSOA, declarada sem relacionamento-chave, podem ser recuperadas por nome, sobrenome e data de nascimento;

3) Classes de instâncias absolutamente iguais. Não são declarados relacionamentos-instância. Propriedades das instâncias são fatoradas por relacionamentos_classe. Por exemplo, numa classe de pregos, o formato, peso e número total de pregos são declarados como relacionamentos-classe;

4) Classes cujas instâncias são identificadas por domínios, tais como os tipos primitivos (inteiros, reais, strings, booleano, etc). Por exemplo, a classe NOME é formada por objetos que são strings.

As classes são organizadas numa estrutura de árvore hierárquica. Os relacionamentos hierárquicos entre classes podem ser considerados abstrações, no sentido de que alguns dos

atributos das subclasses são suprimidos na formação de classes de objetos mais gerais. Os relacionamentos hierárquicos são representados pelos conceitos de generalização, agregação e agrupamento.

[A] Generalização ==> é representada pelo predicado é-um(A,B), significando que A é uma subclasse de B. O conceito de generalização é obtido aplicando um papel a uma classe. Por meio deste papel, a classe é dividida em várias subclasses. O critério de classificação dos objetos pode ser dado por um predicado, uma enumeração ou um índice. As subclasses podem ser disjuntas ou não. Alguns membros da superclasse podem pertencer a nenhuma das subclasses. Os relacionamentos e métodos da superclasse são herdados pelas subclasses.

Uma classe que descreve propriedades comuns de várias classes mais específicas é uma generalização. O inverso é chamado especialização e é guiado por um **papel** ou critério de especialização. Por exemplo, podemos aplicar diversos papéis a uma classe PESSOA e obter

sexo(PESSOA) = MASCULINO, FEMININO

idade(PESSOA) = JOVEM, IDOSO

condição_trabalho(PESSOA) = EMPREGADO, ESTUDANTE, APOSENTADO

Na primeira especialização, as subclasses são disjuntas e cobrem a classe genérica. No último exemplo, ESTUDANTE e EMPREGADO não precisam ser disjuntas e crianças são pessoas que não ocorrem em nenhuma das subclasses.

A figura 3.3. mostra a representação gráfica da

generalização.

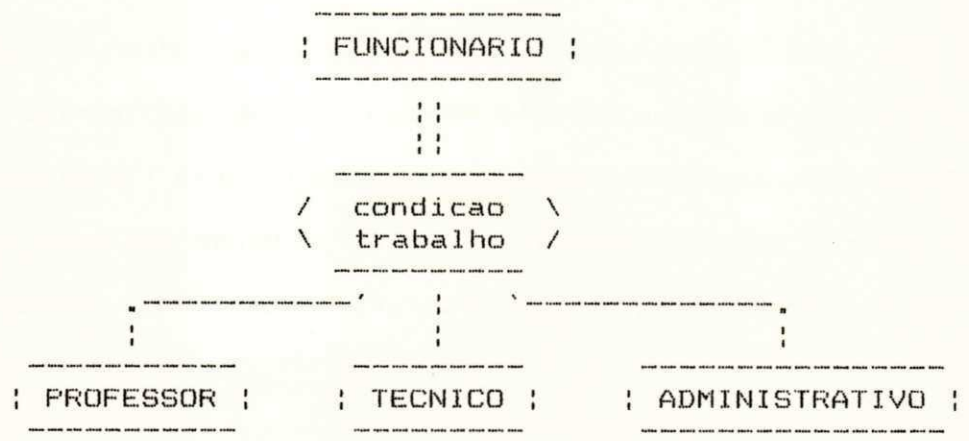


Figura 3.3 - Representação gráfica da generalização

[B] Agregação ==> é representada pelo predicado é-parte(A,B), i.e., A é um componente da classe agregada B. Este conceito permite definir os objetos de uma classe como elementos pertencentes ao produto cartesiano dos membros de outras classes. Os objetos da superclasse são formados pela composição de objetos das subclasses. Por exemplo, cada objeto da classe CORPO-HUMANO é constituido de objetos das classes CABEÇA, TRONCO e MEMBROS. A figura 3.4. ilustra a representação gráfica da agregação.

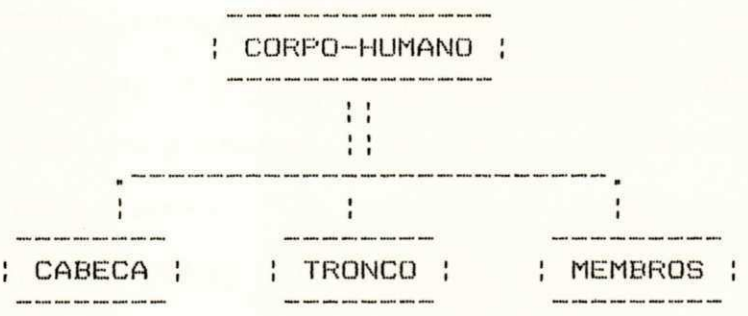


Figura 3.4 - Representação gráfica da agregação

[C] Agrupamento ==> é representado pelo predicado é-elemento(A,B), i.e., os membros de B são conjuntos de membros de A. Esta abstração possibilita a definição dos objetos de uma classe como conjuntos de instâncias da subclasse. Ou, por outro lado, as instâncias de uma classe são agrupadas e esses grupos são considerados como objetos de uma classe superior. Por exemplo, grupos de objetos da classe ARVORE formam instâncias da classe FLORESTA. A figura 3.5. mostra a representação gráfica do agrupamento.

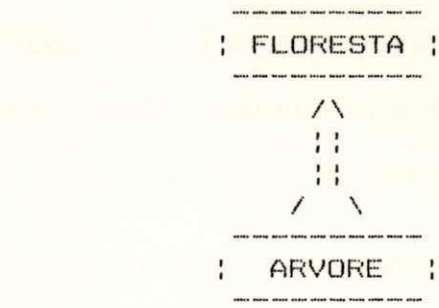


Figura 3.5 - Representação gráfica do agrupamento

No TOM, consideramos herança também para agregação e agrupamento. Na generalização a herança é automática pois ocorre uma reconsideração do mesmo objeto do mundo real num outro nível. Nas outras duas abstrações, novos objetos compostos são introduzidos. Devido à interrelação estreita e física entre o novo objeto e seus componentes, aplica-se uma herança seletiva. Identificamos três casos:

1) Herança Direta - relacionamentos como "tem_cor" ou "localização", e métodos como "move", podem ser herdados pelos objetos de nível inferior sem modificações;

2) Herança Computada - algumas propriedades não podem ser herdadas diretamente. Uma computação deve ser executada. Por exemplo, o "peso" do objeto composto pode ser a soma dos pesos dos componentes. Neste caso, a herança computada é ascendente;

3) Nenhuma herança - existem relacionamentos e métodos que só se aplicam aos objetos compostos. Por exemplo, o relacionamento **tem_placa** se aplica somente à classe CARRO e não aos seus componentes.

Como já foi visto, um dos fundamentos do modelo TOM é a modelagem do tempo. O tempo é modelado através do relacionamento temporal pré-pós, das classes com tempo ("with time") e dos relacionamentos com valores anteriores ("with old values").

[A] O relacionamento temporal pré-pós

Existem casos em que um objeto, ao ser eliminado de uma classe, passa a pertencer a uma outra classe de objetos. Inversamente, ocorrem casos onde um objeto incluido em uma classe já havia pertencido a uma classe de objetos imediatamente anterior. Esse relacionamento mútuo entre duas classes é denominado, no TOM, relação pré-pós e denotada por

C1 >------> C2

diz-se que C1 é a pré-classe e C2, a pós-classe. Se todos os objetos que saem de C1 vão para C2, o relacionamento é dito pré-exclusivo/pós-simples e denotado por

C1 >>-----> C2

Se todos os objetos que estão em C2 vieram de C1, então

a relação é dita pré-simples/pós-exclusiva e denotada por

C1 >------>> C2

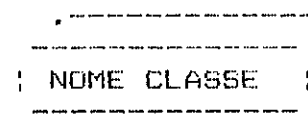
O relacionamento pode ainda ser pré-exclusivo/pós-exclusivo, i.e., todos os objetos que saem de C1 vão para C2 e todos os objetos que estão em C2 vieram de C1.

Quando um objeto membro de uma classe passa a pertencer a uma outra classe, os relacionamentos-membro associados são automaticamente transferidos para a nova classe.

[B] Classes com tempo ("with time")

O parâmetro "with time" de uma classe de objetos especifica que os objetos removidos daquela classe permanecerão com uma indicação do tempo de duração histórico do objeto na classe. O parâmetro "lifetime" indica que os objetos serão efetivamente removidos da classe após o intervalo de tempo especificado. Isto evita que as classes cresçam indefinidamente.

Classes com tempo são representadas graficamente por



[C] Relacionamentos com valores anteriores ("with old values")

O parâmetro "with n old values" especifica que os valores dos relacionamentos alterados serão mantidos no banco de dados para efeito de histórico, juntamente com a indicação das datas de início e término da existência dos valores anteriores do relacionamento.

VERSIONAMENTO

Um objeto versionado em TOM divide-se em duas partes. Uma parte genérica, contendo todas as informações invariantes ao longo das versões do objeto. A outra parte é composta de todas as versões individuais do objeto. Por isso, um objeto versionado se decompõe em um objeto-g cujas propriedades são descritas em uma classe-g e vários objetos-v. Todos os objetos-v estão relacionados entre si por relacionamentos especiais e formam um grafo aciclico, chamado **grafo de versão.** Além disso, eles estão relacionados a um único objeto-g, pela relação "versão_de".

Um objeto versionado tem sua evolução representada por seu grafo de versão. Os arcos entre objetos-v representam o progresso do objeto nessa evolução. O tipo de arco mais simples é a relação **seguinte**, representando uma sequência cronológica única de uma versão para outra. O segundo tipo de arco surge quando temos diversas versões seguintes mutuamente exclusivas, refletindo tentativas alternativas de construção do objeto; esta é a relação **alternativa_de.** Se todos os objetos-v seguintes são variantes válidas do mesmo objeto, nós temos uma relação **variante_de.** Finalmente, em muitos ambientes é necessário decompor um objeto em partes que são desenvolvidas independentemente; esta é a relação **parte_de.**

Resumindo, as formas de versionamento têm as seguintes propriedades:

seguinte: único, total

alternativa_de: múltiplo, total, mutuamente exclusivo

variante_de: múltiplo, total, não-exclusivo

parte_de: múltiplo, parcial, não-exclusivo

Os aspectos dinâmicos do modelo TOM são modelados pelo conceito de métodos. Os objetos do banco de dados só podem ser atualizados por meio de operações pré-definidas, sujeitas a pré-condições. Um dos objetivos principais nas operações de atualização do banco de dados é a manutenção da integridade conceitual. Por isso, o TOM incorpora conceitos de pré-condição e pós-condição sobre métodos, assegurando que os métodos só serão executados se as pré-condições para tal forem satisfeitas. E ainda, ao final da operação as pós-condições devem ser satisfeitas.

Além das pré-condições, o TOM apresenta um sistema de efeitos colaterais, eventos e "triggers". Os efeitos colaterais constituem um conjunto de operações colaterais que precisam ser executadas junto com as operações primitivas do TOM, a fim de manter a integridade do esquema conceitual. O sistema de eventos e "triggers" é uma extensão do mecanismo de pré-condição ("events") para a realização de operações ("triggers") que dependem, entre outras condições, de um tempo adequado para sua execução (clock). Podemos associar a uma classe de objetos, regras evento/"trigger" da forma ON <evento> WHEN <condição> DO <ação>. Maiores detalhes sobre o sistema de efeitos colaterais, eventos e "triggers" [FIL87][SCHI82A] podem ser encontrados no capítulo 4.

As operações primitivas permitem a criação, eliminação

ou movimentação de objetos e possibilitam estabelecer, remover ou atualizar relacionamentos. Os predicados e operações primitivas são listados abaixo.

Predicados Primitivos:

a) e in C [at time]

Verifica se o objeto e pertence à classe C num determinado intervalo de tempo.

b) is_rel(e1,r,e2) [at time]

Verifica se os objetos e1 e e2 estão relacionados por r num determinado intervalo de tempo.

c) is_part(e1,e2)

Verifica se o objeto e1 é componente do objeto agregado e2.

d) is_elem(e1,e2)

Verifica se o objeto e1 é um elemento do objeto de grupo e2.

e) is_a(C1,C2)

Verifica se a classe C1 é uma subclasse de C2.

Os predicados primitivos não citados aqui podem ser encontrados em [FIL87].

Operações Primitivas:

(a) C create (e) {at time}

Cria o objeto e na classe C a partir do

instante especificado.

(b) e delete [(C)] {at time}

Elimina o objeto e de sua classe C, a partir do instante especificado.

Como resultado de um processo de simplificação sintática, surgem as demais operações primitivas: establish, remove, move e update:

(c) r establish (<e1,e2>)

Equivalente à execução de: Cr create (<e1,e2>).

Estabelece o relacionamento r entre os objetos e1 e e2 (relacionamento membro-a-membro). Cr é a classe de relacionamentos r.

r establish (<C,e>)

Equivale à execução de: Cr create (<C,e>).

Estabelece o relacionamento r entre a classe C e o objeto e (relacionamento classe-a-membro). Cr é a classe de relacionamentos r.

(d) <e1,r,e2> remove

Equivale à execução de: <e1,e2> delete(Cr).

Remove o relacionamento r entre os objetos e1 e e2. Cr é a classe de relacionamentos r.

<C,r,e> remove

Equivalente à execução de: <C,e> delete(Cr).

Remove o relacionamento r entre a classe C e o

objeto e. Cr é a classe de relacionamentos r.

(e) e move (C1,C2)

Equivalente à execução das duas operações:

e delete(C1)

C2 create (e)

(f) e1 update (r = e3)

Equivalente à execução das duas operações:

<e1,r,e2> remove

r establish (<e1,e3>)

4. Especificação Axiomática do Modelo TOM

4.1. Introdução

A única primitiva básica é o **objeto**. O modelo TOM mapeia objetos do mundo real para objetos do esquema conceitual; tipos de objetos para classes; propriedades para relacionamentos; processos para métodos e ocorrências para eventos.

Uma **classe** é uma quádrupla

$$C = \langle Ic, Sc, Mc, |C| \rangle$$

onde **Ic** é a identificação de C, composta pelo nome da classe Nc, e outras informações eventuais tais como uma descrição informal da classe;

Sc (Structure c) é o conjunto de estruturas de dados de C, composta de relacionamentos-classe e relacionamentos-instância;

Mc é o conjunto de métodos de C, formado por métodos-classe e métodos-instância;

|C| (Extent c) é o conjunto de instâncias de C.

A seguir descrevemos axiomaticamente a semântica do TOM. Na seção 4.2. são apresentados os axiomas estáticos; na seção 4.3., os axiomas de tempo e versionamento; na seção 4.4., os axiomas dinâmicos; e, finalmente, na seção 4.5., os efeitos colaterais.

4.2. Axiomas Estáticos

Nesta seção nós consideramos apenas os aspectos

estáticos do modelo. O conteúdo das classes varia com o tempo, por isso a representação correta de uma classe C é Ct, i.e., "classe C no instante t" e C é a família de todas as Ct. O primeiro axioma é:

A1: Nc = Nd ==> C = D (classes distintas devem ter nomes distintos)

No que se segue, usaremos C para denotar C, |C| ou Nc. Portanto, nós sempre escreveremos e ∈ C ao invés de e ∈ |C|.

Dadas duas classes C e D, um **relacionamento** r de C em D é um sistema

r = < Nr, Rr, minr, maxr >

onde Nr é o nome de r;

Rr ⊂ |C| x |D|;

minr é um inteiro positivo representando a cardinalidade mínima;

maxr é um inteiro positivo ou um símbolo especial denotado por $\underline{*}$, representando a cardinalidade máxima do relacionamento r.

Daqui por diante usaremos r, Nr, e Rr indistintamente para denotar o mesmo conceito. Um relacionamento também pode ser denotado por r(min,max) ou, se nós queremos identificar as classes relacionadas, CrD. Se CrD é um relacionamento e c é uma instância de C, então o predicado r(c) representa o conjunto de todas as instâncias de D relacionadas a c, i.e.,

r(c) = { d ∈ D / <c,d> ∈ Rr }

e o predicado r(c,d) é satisfeito, s.s.s. <c,d> ∈ Rr. Se C é uma classe de objetos, então instance_relationships(C) representa o conjunto de todos os relacionamentos-instância de C, i.e.,

instance_relationships(C) = { r / r ∈ Sc e r é um relacionamento-instância }

E da mesma forma para class_relationships:

class_relationships(C) = { r / r ∈ Sc e r é um relacionamento-classe }

Os seguintes axiomas devem se aplicar:

A2: (CrD ∧ CsE ∧ Nr = Ns) ==> r = s

(uma classe não pode ter dois relacionamentos com o mesmo nome)

Para assegurar que uma classe não terá dois relacionamentos com o mesmo nome, precisamos de uma pré-condição ao definir um relacionamento **r** em uma determinada classe **A**. Essa pré-condição especifica que não pode existir um relacionamento com o mesmo nome do novo relacionamento **r**, na classe em questão.

Pré-condições são geralmente definidas no contexto de um método. No nosso caso, existe um método create(novorel) para criar um relacionamento entre duas classes.

Implementamos a pré-condição para representar o axioma A2 em duas etapas. Primeiro selecionamos todos os nomes de relacionamentos da metaclasse RELATIONSHIP que estão definidos na classe A, atribuindo o conjunto resultante à variável IR. Em seguida, comparamos o nome do novo relacionamento **r** a ser incluído na classe **A** com o conjunto **IR** de nomes de

relacionamentos já definidos nesta classe. Se não houver um relacionamento já definido com o mesmo nome de r, então r será efetivamente incluído na metaclasse RELATIONSHIP.

A3: maxr <> * ==> minr <= maxr

A4: c ∈ |C| ==> (#{ <c,d> / <c,d> ∈ Rr } >= minr ∧
(maxr <> * ==> #{<c,d>/<c,d> ∈ Rr} <= maxr))

A5: r(c,d) ==> ∃ C,D (c ∈ C ∧ d ∈ D ∧ CrD)
(dois objetos estão relacionados somente se as classes correspondentes estão relacionadas)

Todo relacionamento tem seu inverso

A6: CrD ==> ∃ s (DsC ∧ (r(c,d) <=> s(d,c)))

a relação s é também denotada por r^{-1}. Dependendo das cardinalidades máximas de uma relação e seus inversos, as seguintes caracterizações são obtidas:

Se maxr = $maxr^{-1}$ = 1
então r é 1:1

Se maxr > 1 e $maxr^{-1}$ = 1
então r é 1:n

Se maxr = 1 e $maxr^{-1}$ > 1
então r é n:1

Se maxr > 1 e $maxr^{-1}$ > 1
então r é n:n

Agora passaremos a descrever as estruturas hierárquicas de generalização, agregação e agrupamento.

Antes de descrever a semântica da hierarquia de generalização/especialização, definimos um **papel** como um predicado disjuntivo

$$p(e) = p1(e) \vee \ldots \vee pn(e)$$

onde $\underline{e}$ é um objeto de uma classe genérica G tal que pi(e) é verdadeiro s.s.s. $\underline{e}$ é uma instância da subclasse Ci.

Uma **generalização** G de classes C1, ..., Cn pelo papel p é dada por

A7: para i = 1,2,...,n é_um(Ci,G,p) <==>

$$((e \in G \wedge pi(e)) <==> e \in Ci)$$

Isso caracteriza as classes Ci como subclasses de G com $|Ci| \subset |G|$. Um papel aplicado a uma classe D é **disjuntivo** se as subclasses são disjuntas:

A8: para 1 <= i, j <= n $\wedge$ i <> j

disjuntivo(D, C1, ..., Cn, p) <==> (pi(e) ==> $\sim$ pj(e))

Se cada objeto da classe genérica está em pelo menos uma subclasse, o papel é dito "covering":

A9: covering(G, C1, ..., Cn, p) <==> (e $\in$ G ==> $\exists$ i pi(e))

Como consequência das definições anteriores, temos:

i) Se p é disjuntivo então $Ci \cap Cj = \emptyset$ para i <> j

ii) Se p é "covering" então $\bigcup_{i=1}^{n} Ci = G$

A segunda estrutura hierárquica é obtida pela agregação de duas ou mais classes de objetos para formar uma classe composta. Uma classe A é uma **agregação** de classes C1,...,Cn se

A10: agregado(A,C1,...,Cn) <==> |A| $\subset$ |C1| x ... x |Cn|

Como já foi visto anteriormente, a expressão |A| (Extent-A) representa o conjunto de instâncias de A. A é uma **agregação de C1,...,Cn pelos relacionamentos** rij s.s.s. exatamente os objetos relacionados por rij estão na classe agregada

A11: agregado_r(A,C1,...,Cn,{rij}) <==>

i) <c1,...,cn> $\in$ A <==> (<c1,...,cn> $\in$ C1 x ...x Cn $\wedge$
(r $\in$ {rij} ==> $\exists$ 1 <= k, l <= n r(ck,cl)))

(os componentes das instâncias de A devem estar nas classes componentes e relacionados entre si)

ii) CrD ==> (r $\in$ {rij} ==> C,D $\in$ {C1,...,Cn})

iii) todas as classes componentes são transitivamente relacionadas:

para i,j = 1,...,n $\exists$ k1,...,km

(Ci ri k1, Ck1 rk1 k2 ... rkm j Cj) $\wedge$

Ck1,...,Ckm $\in$ {C1,...,Cn}

Uma classe agregada A é uma **redefinição** se suas classes componentes C1', ..., Cn' são redefinições das classes componentes C1, ..., Cn da estrutura de agregação herdada das classes superiores, na hierarquia:

A12: agregação_redefinição(A,C1',...,Cn') <==>

$$(\exists\, G\ (\text{é\_um}(A,G,p)) \ \wedge\ \text{agregado}(G,C1,\ldots,Cn) \ \wedge\ \text{para } i = 1,\ldots,n\ ,\quad |Ci| \supseteq |Ci'|)$$

|Ci| (Extent Ci) representa o conjunto de instâncias da classe Ci.

A é uma agregação de C1,...,Cn com herança direta, s.s.s., os objetos das classes componentes herdam os relacionamentos e métodos da classe agregada, diretamente. O relacionamento r é herdado diretamente pelos objetos componentes, s.s.s.

A13: agg_herança (A,C1,...,Cn,r) <==>

(agregado(A,C1,...Cn) ∧ (a ∈ A ∧ a r b ==> (is_part(ai,a) ==> ai r b)))

A é uma agregação de C1,...,Cn com herança computada se, e somente se, os objetos da classe agregada herdam propriedades computadas das classes componentes. O relacionamento r tem herança computada através da função θ, s.s.s.

A14: agg_herança_comp (A,C1,...,Cn,r,θ) <==>

(agregado(A,C1,...,Cn) ∧

(a ∈ A ∧ a r b ==> (is_part(ai,a) ∧

ai r bi, i=1,...,n ==> $b = \theta^{n}_{i=1}(bi)$))))

A terceira e última estrutura hierárquica é obtida pelo agrupamento de objetos de uma classe para formar objetos de uma classe superior. Uma classe G é um **agrupamento** de uma outra classe C pelo predicado p, se seus elementos são conjuntos de

UFPb/BIBLIOTECA/PRAI

elementos de C e p é verdadeiro entre elementos e grupos:

A15: é_elem(C,G,p) <==> $G \subset P(C)$ $\wedge$
((g $\in$ G $\wedge$ x $\in$ C $\wedge$ x $\in$ g) ==> p(x,g)) $\wedge$
(p(x,g) ==> (x $\in$ C <==> x $\in$ g))

Um agrupamento é **disjuntivo** se cada objeto ocorre em exatamente um grupo.

A16: disjuntivo(C,G,p) <==> (é_elem(C,G,p) $\wedge$
(g1,g2 $\in$ G ==> g1 $\cap$ g2 = $\emptyset$))

Um agrupamento G de C é "**covering**" se todos os objetos da classe elemento ocorrem em pelo menos um grupo:

A17: covering(C,G,p) <==> (é_elem(c,G,p) $\wedge$ $\cup$ C = G)

Um agrupamento G de C é **ordenado** ("ordered") se os objetos estão dispostos numa sequência f: $|N_n^*$ --> g dentro de cada grupo g:

A18: ordered(C,G,p) <==> (é_elem(C,G,p) $\wedge$
(g $\in$ G ==> $\exists$ f: $|N_n^*$ --> g))

4.3. Axiomas de Tempo e Versionamento

O tempo é considerado como uma classe T de tuplas t = (t1:u1, ..., tn:un) tal que:

i) há um conjunto de constantes p2,...pn chamadas períodos;

ii) t1,...,tn são inteiros positivos tal que ti < pi;

iii) u1,..., un são strings, chamadas unidades (tais como anos, meses, dias, horas, minutos, segundos);

iv) para i = 2,...,n pi:ui = 1:ui+1 (e.g. 60:segundos = 1:minuto) e para i = 1, p1 = ∞

A menor unidade un é chamada **granularidade** dos pontos de tempo (e.g. segundos). A eliminação de unidades inferiores implica uma granularidade maior (e.g. minutos). A eliminação de unidades superiores (e.g. anos, p1 = ∞) implica em pontos de tempo periódicos.

Há um ponto especial no tempo que reflete o "momento presente" e que denominamos "**now**". O "now" é uma representação do instante de tempo do mundo real através de um relógio interno (com calendário). O valor do "momento presente" é retornado pela função "now".

T tem uma relação de ordenação natural < de pontos de tempo dada pelos conceitos de "antes" e "depois". Uma classe I é uma **classe de intervalos de tempo** se ela é um agrupamento de uma classe de tempo T tal que cada ponto de tempo entre dois pontos num grupo de tempo (ou intervalo) está neste grupo:

A19: intervalo(I,T) <==> (p,r ∈ T ∧ p,r ∈ i ∈ I) ==>
(p < q < r ==> q ∈ i)

O limite inferior de um intervalo de tempo é chamado o ponto "**begin**" e o limite superior, o ponto "**end**", e se t = **begin** e s = **end**, então i = (t,s). O intervalo (t,now) significa "de t até agora".

Uma **classe com tempo** é uma classe agregada C' = C × I de uma classe C e uma classe de intervalos de tempo I, tal que

A20: timed(C',C,I) <==> intervalo(I,T) ∧ agregado(C',C,I) ∧
(<c,i1>,<c,i2> ∈ C' ==> (i1 = i2 ∨ i1 ∩ i2 = ∅))

A21: timed(C',C,I) ==> (x ∈ C <==> ∃ t (<x,(t,now)> ∈ C'))

Um par ct = <c,t> ∈ C' é chamado timed-object(ct) s.s.s.,

A22: timed-object(ct) <==> (begin(ct) = begin(t) ∧
end(ct) = end(t))

Uma **classe temporal** é uma classe agregada C" = C'x I de uma classe com tempo C' = C x I, tal que

A23: temporal(C",C',I) <==> ∃ C [timed(C',C,I)]

Quando o valor de um relacionamento é alterado (e.g. "trabalha_depart" entre as classes EMPREGADO e DEPARTAMENTO) pode ser necessário o armazenamento dos departamentos onde o empregado já trabalhou. Uma **relação com valores anteriores** de C em D é um sistema

r' = < N, R', min, max >

tal que

i) <N,R'> é uma classe com R' ⊂ |C| x |D| x I
onde I é uma classe de intervalos de tempo;

ii) <c,d,i1>, <c,d,i2> ∈ R' ==> (i1 = i2 ∨ i1 ∩ i2 = ∅);

iii) para cada ponto de tempo t ∈ i ∈ I, se Rt é a projeção_t de R' sobre |C| x |D|, i.e., Rt = {s ∈ |C| x |D| / ∃i(t ∈ i ∨ <s,i> ∈ R')} então rt = < N, Rt, min, max > é uma relação de C em D;

iv) ∀ c ∈ C ∃to ∀ t (t >= to ∧ t <= now ==> ∃ i,d (t ∈ i ∧ <c,d,i> ∈ R')).

Baseado nos conceitos acima, determinam-se os axiomas:

A24: old(r') <==> condições i) a iv) são satisfeitas

A25: old(r") <==> old(r') e condições i) a iv) ocorrem sobre r"

Para evitar que o conteúdo das classes cresça indefinidamente pelo uso de classes com tempo, o conceito de **"lifetime"** é introduzido. Este parâmetro especifica um tempo de vida para os objetos de uma classe, após o qual os objetos históricos são removidos da classe.

Uma **relação pré-pós** entre classes (denotada por C >--> D), estabelece que os objetos removidos da classe C podem ser inseridos em D e os objetos inseridos em D podem ter se originado de C. C é uma **pré-classe** de D e D, uma **pós-classe** de C. Uma relação pré-pós é **exclusiva** quando os objetos removidos da pré-classe são obrigatoriamente inseridos na pós-classe, e os objetos inseridos na pós-classe obrigatoriamente se originam da pré-classe. A relação pré-pós exclusiva é denotada por C >>-->> D. Por exemplo:

CARRO_FABRICANTE >>--> CARRO_EM_USO >-->> CARRO_DESTRUIDO

significa que todo carro fabricado entra em uso, mas nem todo carro em uso veio diretamente do fabricante. Todo carro destruído um dia esteve em uso, mas nem todo carro usado é destruído. Formalmente, temos os seguintes axiomas:

A26: excl_pre(C,D) <==> (~ x ∈ D ∧ o(x ∈ D) ==> x ∈ C)

A27: excl_pos(C,D) <==> (x ∈ C ∧ o(~ x ∈ C) ==> o(x ∈ D))

Onde o operador temporal o(p) significa que "no próximo estado p é verdadeiro".

VERSIONAMENTO

Sejam v0, v1, v2 versões de um mesmo objeto, tal que timed-object(vi), i=0,1,2.

A relação de evolução cronológica de uma versão **v1** para uma versão **v2** é dada pelo predicado **seguinte**

A28: seguinte(v1,v2) <==> (End(v1) = Begin(v2) $\wedge$
g-object(v1) = g-object(v2))

A relação **alternativa-de** entre duas versões **v1** e **v2** é dada por

A29: alternativa_de(v1,v2) <==> (Begin(v2) >= End(v1) $\wedge$
$\exists$ v (seguinte(v,v1) $\wedge$ seguinte(v,v2)))

Duas versões v1 e v2 são **variantes** se elas ocorrem no mesmo instante e diferem de alguma forma pelo seu conteúdo

A30: variante_de(v1,v2) <==> $\exists$ v (Sv $\subseteq$ Sv1 $\wedge$ Sv $\subseteq$ Sv2 $\wedge$
Sv1 <> Sv2 $\wedge$ seguinte(v,v1)
$\wedge$ seguinte(v,v2))

onde Sv (Structure v) é o conjunto de estruturas de v (conjunto de instance_relationships e instance_methods).

Versões distintas v1 e v2 são **partes** somente se são componentes distintas do mesmo objeto

A31: parte_de_v(v1,v2) <==> $\exists$ v (Sv $\subseteq$ Sv1 U Sv2 $\wedge$
Sv1 $\cap$ Sv2 = $\emptyset$ $\wedge$ seguinte(v,v1) $\wedge$
seguinte(v,v2) $\wedge$ agregado(v,v1,v2))

Definição:

$$\text{descendente\_de}(v,v') \overset{\text{def.}}{\Longleftrightarrow} (\text{seguinte}(v,v0) \Longrightarrow v' = v0 \vee \text{descendente\_de}(v0,v'))$$

A32: $\text{integrado\_de}(v,v1,v2) \Longleftrightarrow (Sv \subseteq Sv1 \cup Sv2 \wedge Sv1 \cap Sv2 = \emptyset \wedge \text{seguinte}(v1,v) \wedge \text{seguinte}(v2,v) \wedge \exists\, v0\ (\text{descendente\_de}(v0,v1) \wedge \text{descendente\_de}(v0,v2)))$

4.4. Axiomas Dinâmicos

Nas seções anteriores foi apresentada a semântica das hierarquias do TOM e dos seus conceitos de tempo e versionamento. Nesta seção, definimos a semântica das operações sobre os objetos do banco de dados, também conhecidas como **métodos.**

Consideremos a representação de todos os estados do universo de discurso no passado, presente e futuro, ao qual damos o nome de DUD (Descrição do Universo de Discurso). DUD é composto de:

UO = { o / o é um objeto do DUD }

UC = { C / C é uma classe no DUD }

UR = { r / r é um relacionamento no DUD }

O universo de objetos é uma união de conjuntos disjuntos, chamados **tipos de objetos.** Para um objeto $\underline{o}$ o tipo a que ele pertence é denotado por To, e dizemos que $\underline{o}$ é do tipo To.

UC é composto de subconjuntos disjuntos chamados **metaclasses,** tal que há uma bijeção entre tipos de objetos e metaclasses. Se MC $\subset$ P(UC) é o conjunto de todas as metaclasses e TO, o conjunto de todos os tipos de objetos, temos os

mapeamentos

rep: TO -> MC e int: MC -> TO

Seja Met o conjunto de todos os nomes de método e si i=1,...,n, os conjuntos de todos os nomes de argumentos.

Uma "**signature**" para um método m, é uma função que mapeia

s1 x s2 x ... x sn --> s

onde s1,s2,...,sn são os nomes dos parâmetros de entrada de m, e s é o valor resultante da operação. Um método m é um par m = (n, δ) onde n é um nome de método e δ é uma "signature"[LRV89]. Uma expressão de mensagem é uma tripla exp = (o,n, δ), onde o é um identificador de objeto, n é o nome de um método e δ é uma "signature".

A identificação do corpo do método a ser executado depende da classe do objeto receptor e do nome do método, e pode ser vista conceitualmente como um mapeamento de

UO x Met --> Impl

onde Impl = UO x $Valor^{*}$ --> Valor é o conjunto de implementações de métodos. Uma implementação de método modela um método definido numa classe de objetos como uma função que recebe o objeto receptor e os parâmetros reais como argumentos, retorna um valor, e geralmente atualiza o estado do objeto.[KLA90]

O objeto receptor e o nome do método identificam a implementação do corpo do método.

As operações básicas do TOM são a criação de uma nova instância de uma classe ou metaclasse, a criação de uma versão, e

a eliminação de uma instância de uma classe ou metaclasse. Se T e S são tipos de objetos e M = rep(T), uma metaclasse correspondente, então

1) create crea: T x M --> M

(o , C) -> C U {o}

1.1) createVersion creaV: T x S --> M

(g , v) -> V_CLASS U {v}

2) delete del: T x M --> M

(o , C) -> C - {o}

Operações adicionais establish, remove, move, update podem ser definidas como combinações das operações básicas ou diretamente:

3) establish est: T x S x UR --> UR

(o , p , r) --> r U {<o,p>}

4) remove rem: T x S x UR --> UR

(o , p , r) --> r - {<o,p>}

5) move mov: T x M x M --> M x M

(o , C , D) -> (C', D') onde C' = C - {o}

e D' = D U {o}

6) update upd: T x S x S x UR --> UR

(o , p , q , r) -> (r - {<o,p>}) U {<o,q>}

Duas operações especiais para hierarquias de agrupamento são necessárias: g_insert(o,g) inclui o como um novo elemento do grupo g e g_delete(o,g) elimina o de g. O efeito dessas funções é o mesmo de create e delete, apenas os domínios são diferentes.

A execução de uma operação primitiva implica na

execução de outras operações implícitas sobre as estruturas hierárquicas, necessárias para manter a integridade semântica do esquema conceitual. Tais operações implícitas são denominadas **efeitos colaterais do esquema.** Numa aplicação concreta, outros efeitos colaterais podem ser explicitamente definidos por meio de eventos e "triggers" e são chamados **efeitos colaterais do usuário.** Os efeitos colaterais do esquema, em conjunto com o conceito de operações primitivas, garantem a integridade do banco de dados, liberando o usuário da tarefa de definir explicitamente um conjunto de restrições de integridade.

Os axiomas para as regras dinâmicas são escritos em lógica dinâmica com dois tipos de fórmulas:

1) p ¦- [op] => q para os axiomas dinâmicos, significa que "num estado em que p é verdadeiro, op é permitido somente se q é verdadeiro";

2) p ¦- [op1] => [op2] para os efeitos colaterais, significa que "num estado com p verdadeiro, a execução de op1 implica a execução de op2".

O operador temporal o(p) ("no próximo estado p é verdadeiro") será também necessário para alguns efeitos colaterais.

As condições gerais que regem a execução das operações primitivas, chamadas de axiomas dinâmicos, são listadas abaixo:

AD1: ¦- [establish(x,y,r)] => ∃ C,D (x ∈ C ∧ y ∈ D ∧ CrD ∧
(maxr = * v #{<x,z> / r(x,z)} < maxr))
(establish deve conservar a cardinalidade máxima)

AD2: ¦- [remove(x,y,r)] => #{ <x,z> / r(x,z)} > minr

(remove deve conservar a cardinalidade minima)

AD3: é_um(C,D,p) ¦- [create(x,C)] => pc(x)

(create deve conservar o papel)

AD4: covering(D,C1,...,Cn) ^

(é_um(Ci,D,p) => ~ x ∈ Ci)

¦- [create(x,D)] => ∃ i(pci(x))

(numa generalização "covering" não é permitido que um objeto pertença apenas à classe genérica)

AD5: timed(C',C,I) ¦- [create(x,C)] => ~ <x,(t,now)> ∈ C'

(numa classe com tempo não é permitida a criação de uma instância já presente)

AD6: é_elem(C,G,p) ¦- [g_insert(x,g)] => p(x,g)

(elementos de grupo devem obedecer ao predicado do agrupamento)

Estes axiomas estabelecem que as operações podem ser executadas somente se determinadas condições forem satisfeitas.

4.5. Efeitos Colaterais

Antes de apresentar os efeitos colaterais é preciso definir dois predicados sobre objetos

is_part(xi,y) significa "o objeto xi é o i-ésimo componente do objeto agregado y"

agregado_r(y,x1,...,xn,r1,...,rm) significa " o objeto

y é composto de x1,...,xn relacionados por r1,...,rm; neste caso denotamos também por y = <x1,...,xn>"

Os efeitos colaterais dependem da posição hierárquica da classe afetada. A seguir são descritos os efeitos colaterais possíveis em cada uma das três hierarquias de generalização, agregação e agrupamento com seus inversos: especialização, decomposição e dissolução, respectivamente [SCHI83]. Os efeitos colaterais são modelados no metanível por regras evento/"trigger" da forma ON <evento> WHEN <condição> DO <ação>, associadas a metaclasses.

GENERALIZAÇÃO

EC1: é_um(C,D) $\wedge$ ~ in(x,D)

¦- [create(x,C)] ==> [create(x,D)]

(um objeto de uma subclasse deve pertencer à superclasse)

EC2: é_um(C,D) $\wedge$ disjuntivo(D,C1,...,Cn) $\wedge$

$\exists$ i (1 <= i <= n $\wedge$ x $\in$ Ci $\wedge$ C <> Ci)

¦- [create(x,C)] ==> [delete(x,Ci)]

(um create não deve violar uma generalização disjunta)

EC3: é_um(C,D) $\wedge$ covering(D,C1,...,Cn) $\wedge$

(Ci <> C ==> ~ x $\in$ Ci)

¦- [delete(x,C)] ==> [delete(x,D)]

(um delete não deve violar uma generalização "covering")

EC4: x ∈ C ∧ é_um(C,D,p)

¦- ([establish(x,y,r)] ∨

[remove(x,y,r)]) ==>

(pc(x) ∧ o(~ pc(x))) ==> [delete(x,C)]

∧ (~ pc(x) ∧ o(pc(x))) ==> [create(x,C)]

(se, como consequência da alteração de um relacionamento, não é permitido a um objeto permanecer numa subclasse, ele deve ser movido para uma subclasse compatível)

ESPECIALIZAÇÃO

EC5: é_um(C,D,p) ∧ pc(x)

¦- [create(x,D)] ==> [create(x,C)]

(um novo objeto de uma classe genérica deve ser inserido em todas as subclasses compatíveis)

EC6: é_um(C,D) ∧ x ∈ C

¦- [delete(x,D)] ==> [delete(x,C)]

AGREGAÇÃO

EC7: agregado_r(y,x1,...,xn,r1,...,rm) ∧

y ∈ D ∧ (1 <= i <= n ==> xi ∈ Ci)

¦- [delete(xi,Ci)] ==> [delete(y,D)]

(se um objeto agregado ficar sem um de seus componentes, ele deve ser eliminado)

EC8: agregado_r(y,x1,...,xn,r1,...,rm) ∧

y ∈ D ∧ (1 <= i <= n ==> xi ∈ Ci) ∧ rk ∈ {r1,...,rm}

¦- [remove(xi,xj,rk)] ==> [delete(y,D)]

(para uma agregação por relacionamentos, estes relacionamentos devem ser satisfeitos para os objetos agregados)

EC9: agregado_r(D,C1,...,Cn,r1,...,rm) ∧
x1 ∈ C1 ... xn ∈ Cn ∧ ∃rk ∈ {r1,...,rm}
(Ci rk Cj ∧ ~ relacionado(xi,xj,rk))
|- [establish(ci,cj,rk)] ==> [create(<x1,...,xn>,D)]
(este é o inverso de EC8. Se, para um conjunto de objetos, todas as relações de uma agregação são satisfeitas, o objeto agregado correspondente deve ser inserido na classe agregada)

DECOMPOSIÇÃO

EC10: agregado(D,C1,...,Cn)
|- [create(y,D)] ==> (1 <= i <= n ∧ is_part(xi,y) ==>
[create(xi,Ci)])
(as partes de um objeto agregado devem pertencer às classes componentes)

EC11: agregado_r(D,C1,...,Cn,r1,...,rm)
|- [create(y,D)] ==> (1<=i<=n ∧ is_part(xi,y) ==>
[create(xi,Ci)] ∧
(relacionado(Ci,Cj,rk) ∧ rk ∈ {r1,...,rm}
==> [establish(ci,cj,rk)]))
(semelhante a EC10, sendo que os relacionamentos correspondentes devem ser estabelecidos)

AGRUPAMENTO

EC12: é_elem(C,G,p) ∧ g ∈ G ∧ p(x,g)

⊢ [create(x,C)] <==> [g_insert(x,g)]

(Se p(x,g) é satisfeito então x pertence à classe elemento s.s.s. x está em g)

EC13: covering(C,G,p) ∧ ~ ∃g(p(x,g))

⊢ [create(x,C)] ==> [create({x},G)]

∧ p(x,{x})

(por um agrupamento "covering", cada objeto da classe elemento deve estar em pelo menos um grupo)

EC14: é_elem(C,G,p) ∧ x ∈ g

⊢ [delete(x,C)] ==> [g_delete(x,g)]

(vide EC12)

DISSOLUÇÃO

EC15: é_elem(C,G)

⊢ [create(g,G)] ==> (x ∈ g ∧ ~ x ∈ C ==> [create(x,C)])

(os elementos de um grupo devem estar na classe elemento)

EC16: covering(C,G)

⊢ [delete(g,G)] ==> (x ∈ g ∧ ~ ∃h(h ∈ G ∧ h <> g ∧ x ∈ h))

==> [delete(x,C)]

EC17: disjuntivo(C,G)

⊢ [g_insert(x,g)] ==> ∃h ((h <> g ∧ x ∈ h)

==> [g_delete(x,h)])

(se um g_insert viola a propriedade da disjunção, o

objeto é removido dos outros grupos)

Outros efeitos colaterais são:

EC18: relacionado(x,y,r) $\wedge$ x $\in$ C

¦- [delete(x,C)] ==> [remove(x,y,r)]

(um delete provoca a remoção de todos os relacionamentos existentes)

EC19: ~ relacionado(y,x,r^{-1})

¦- [establish(x,y,r)] ==> [establish(y,x,r^{-1})]

(toda relação tem um inverso)

Os efeitos colaterais que se referem aos aspectos de tempo são:

EC20: timed(C',C,I)

¦- [create(x,C)] <==> [create(<x,(t,now)>,C')]

EC21: timed(C',C,I)

¦- [delete(x,C)] <==> (<x,(t,now)> $\in$ C')

==> [delete(<x,(t,now)>,C')] $\wedge$

[create(<x,(t,t2)>,C')]

(numa classe com tempo, um objeto eliminado é movido para o passado)

EC22: old(r') $\wedge$ t = now

¦- [establish(x,y,rt)]

<==> [create(<x,y,(t,now)>,R')]

(EC22 e EC23 se referem a relacionamentos com valores anteriores)

EC23: old(r') ∧ t = now

¦- [remove(x,y,rt)] <==> (<x,y,(to,now)> ∈ R') ==>

[delete(<x,y,(to,now)>,R')] ∧

[create(<x,y,(to,t)>,R')]

EC24: excl_pre(C,D)

¦- [create(x,D)] ==> [delete(x,C)]

EC25: excl_pos(C,D)

¦- [delete(x,C)] ==> [create(x,D)]

Os efeitos colaterais que se referem à agregação com herança são:

EC26: agg_herança(A,A1,...,An,r) ∧ is_part(ai,a),i=1,...,n

¦- update(a,p,q,r) ==> update(ai,p,q,r)

EC27: agg_herança_comp(A,A1,...,An,r,θ) ∧ is_part(ai,a)

¦- update(ai,p,qi,r) ==> update(a,p,q,r) ∧

$q = \theta^{n}_{i=1}\ (qi)$

Os efeitos colaterais que se referem aos aspectos de versionamento são:

EC28: ~ in(g,G_CLASS) ¦- createVersion(g,v)

==> create(g,G_CLASS) ∧ establish(v,g,g_object)

(não pode existir uma versão sem um objeto_g correspondente)

EC29: (1 <= i <= n ==> g = g_object(vi))

¦- delete(g,G_CLASS) ==> delete(vi,V_CLASS)

(ao eliminar um objeto genérico, todas as versões associadas devem ser removidas)

EC30: ¦- next(v1,v2) <==> o(update(v2))

(a alteração da estrutura de uma versão implica a criação de uma versão seguinte com estrutura modificada)

5. Meta-Esquema do Modelo TOM

5.1. Introdução

Os sistemas abertos têm atraído o interesse dos pesquisadores em muitas áreas da ciência da computação. Novos sistemas de gerência de bancos de dados têm sido desenvolvidos para englobar as características dos sistemas abertos. Mas o que é um sistema aberto? Quais são as suas características? Quais são as suas vantagens?

O conceito de sistema aberto parece ter surgido no ambiente de organização de computadores e sistemas operacionais. Um sistema operacional aberto é aquele que pode ser usado em computadores de qualquer fabricante e em qualquer configuração, em contraposição aos sistemas proprietários que são fortemente dependentes do computador de um fabricante específico. As vantagens dos sistemas operacionais abertos são, principalmente, a diminuição do custo de treinamento em um novo sistema operacional, a independência do fabricante do computador e a facilidade de transporte dos sistemas aplicativos para um computador mais poderoso.

A área de bancos de dados vem sofrendo grande influência dos chamados sistemas abertos. Esta influência se reflete particularmente na concepção de um modelo de dados "aberto". Um modelo de dados é aberto quando ele permite que o projetista especifique novos conceitos e abstrações que são incluídos no modelo, tornando possível a modelagem de problemas de propósito especial. Aqui nós distinguimos dois níveis de

projeto ou modelagem: o metanível e o nível de aplicação. No metanível, o projetista do modelo especifica novos conceitos, estruturas e abstrações do modelo, i.e., ele modela o modelo. No nível de aplicação, os conceitos do modelo são usados na representação de um sistema de informação específico.

De acordo com o segundo Modelo de Referência para SGBDs ANSI/SPARC [ANSI86] um sistema de banco de dados é descrito em duas dimensões ortogonais. A dimensão de **ponto-de-vista** engloba três níveis de esquema: externo, conceitual e interno. A outra dimensão, chamada dimensão **intenção-extensão**, se aplica a cada um desses esquemas e determina quatro níveis de descrição. O primeiro é a extensão do banco de dados em si ou **dados de aplicação.** Sua intenção é o **esquema de aplicação** ou dicionário de dados, no segundo nível. Cada esquema de aplicação é uma extensão do **modelo de dados.** A intenção do modelo de dados está no nível mais alto, é o **esquema do modelo de dados**, onde é possível definir e modificar um ou mais modelos de dados. Este esquema do modelo de dados é o **metanível** que caracteriza um ambiente aberto.

O modelo TOM parte do princípio de que existe uma metaclasse pré-definida, denominada TOM_CLASS, integrando um metanível no qual são definidos todos os conceitos do modelo de dados TOM. Estes conceitos são definidos por meio de metaclasses que descrevem sua sintaxe (estrutura) e semântica (comportamento). As metaclasses definidas pelo projetista do modelo representam abstrações, conceitos e estruturas inerentes ao modelo em si. As classes de aplicação são definidas como instâncias das metaclasses definidas pelo projetista do modelo e/ou da metaclasse pré-definida TOM_CLASS. Propriedades

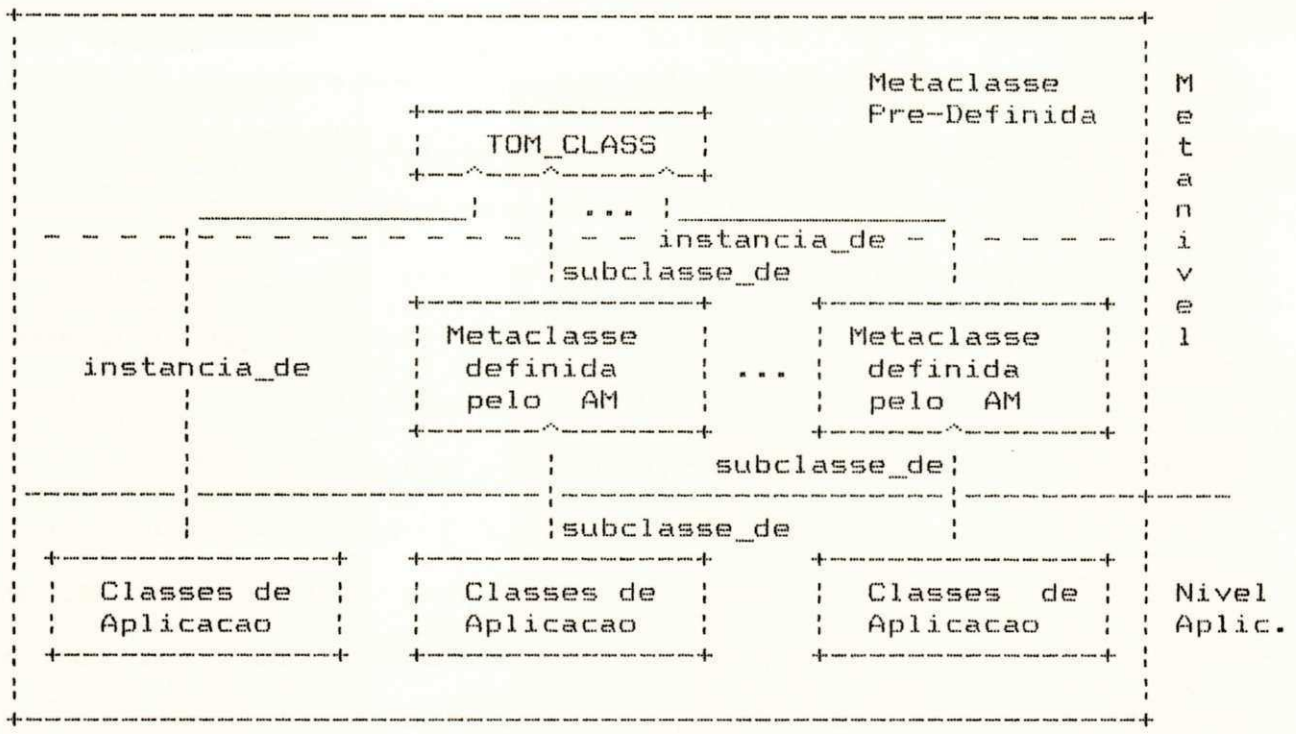


Figura 5.1 - Níveis de abstração do TOM

Nós usamos o sistema de metaclasses para modelar um novo modelo orientado a objetos que inclui relacionamentos temporais, e damos a este modelo o nome de TOM (Temporal Object Model). O novo modelo de dados orientado a objetos fornece os relacionamentos semânticos clássicos entre classes de objetos. As abstrações de agregação, generalização e agrupamento, presentes no modelo THM, também são incluídas no modelo TOM. Estas abstrações hierárquicas possibilitam a definição de uma hierarquia de classes e, portanto, traz embutido um mecanismo de herança.

Como uma característica adicional em um modelo orientado a objetos, o modelo TOM introduz o conceito de relacionamento temporal entre classes de objetos, representado pelo relacionamento temporal pré-pós, relacionamentos com valores

anteriores(histórico) e classes com tempo.

As operações do modelo THM são tratadas, no TOM, como métodos associados a classes de objetos. Esta é uma estratégia necessária para dar ao modelo TOM a característica de encapsulamento. As pré-condições e pós-condições que regem as operações THM são consideradas no TOM como regras de integridade associadas a cada classe de objetos.

Nas seções que se seguem, será introduzida a linguagem usada no metanível. A sintaxe do modelo TOM será descrita em função do sistema de metaclasses acima citado. O formalismo axiomático usado na descrição da semântica do TOM é uma extensão da linguagem formal aplicada à definição da semântica do THM. A base teórica utilizada na descrição da semântica do TOM é o mecanismo de pré-pós condições dos métodos associados às metaclasses e as regras evento/"trigger" da forma ON <evento> WHEN <condição> DO <ação>.

5.2. O Metanível

A definição de modelos de dados se dá no metanível utilizando, basicamente, a linguagem conceitual do TOM. No que se segue, damos uma breve introdução à sintaxe da linguagem conceitual do TOM. Uma listagem completa desta linguagem pode ser encontrada no apêndice.

```
metaclass <nome metaclasse>
   [specialization of <nome metaclasse>]
   [instance of <nome metaclasse>]
   [
    class relationships
       (<nome  rel>  ":" <classe relacionada>
          "(" <card_min>  "," <card_max> ")" )*
```

```
        (pre_class ":" <classe relacionada> [exclusive])*
        (post_class ":" <classe relacionada> [exclusive])*
    ]
    [class methods
        <def.método>+      ]
    [
     instance relationships
        (
         <nome rel> ":" <classe relacionada>
           "(" <card_min> "," <card_max> ")"
         [with <n> (history¦rollback¦temporal) values]
        )+
    ]
    [instance methods
        <def.método>+      ]

    ( keys are ( <lista de chaves>+ ¦ inherited )
    ¦ type <tipo> [ format <especificação> ]

    [<def.papel>*]
    [<def.agrupamento>]
    [<def.agregação>]
    [parameters :  with time [and lifetime <constante> :
                                  <unidade tempo>]      ]
    [on <evento> when <condição> do <ação>]
```

SEMANTICA

specialization of <nome metaclasse>

Declara que a metaclasse sendo definida é uma subclasse de <nome metaclasse>.

instance of <nome metaclasse>

Declara que a metaclasse sendo definida é uma instância de <nome metaclasse>.

class relationships

```
     (<nome  rel>  ":" <classe relacionada>
        "(" <card_min>  "," <card_max> ")" )*
     (pre_class ":" <classe relacionada> [exclusive])*
     (post_class ":" <classe relacionada> [exclusive])*
```

Declara que <nome rel> é um relacionamento existente entre a classe em definição como um todo, e uma instância da

<classe relacionada>. <card min> e <card max> especificam que cada instância da primeira classe está relacionada a no mínimo <card min> e no máximo <card max> instâncias da segunda classe. Pre_class e pos_class declaram a pré-classe e a pós-classe da classe em definição (vide descrição geral do modelo TOM na seção 3.2.).

class methods
<def.método>+

Declara os métodos de classe associados com a classe em definição (vide seção 3.2. para uma descrição mais detalhada de método-classe).

instance relationships
(
<nome rel> ":" <classe relacionada>
"(" <card_min> "," <card_max> ")"
[**with** <n> (**history**|**rollback**|**temporal**) **values**]
)+

Declara que cada instância da classe em definição está relacionada por <nome rel> a uma ou mais instâncias da <classe relacionada>. O parâmetro **with ... values** especifica se os valores dos relacionamentos alterados serão mantidos e a característica dos intervalos de tempo (histórico, rollback ou temporal.

instance methods
<def.método>+

Declara os métodos de instância associados com a classe em definição (vide seção 3.2. para uma descrição mais detalhada de método-instância).

keys are (<lista de chaves>+ ¦ **inherited**)

Lista de chaves é composta por um ou mais relacionamentos que identificam completamente as instâncias da classe.

type <tipo> [**format** <especificação>]

Declara a presente classe como uma classe de domínio.

<def.papel> ::=
with role <nome papel>
gives subclasses <lista de classes> **[explicit]**

Define uma hierarquia de generalização tendo como classe genérica a classe em definição e, como subclasses, <lista de classes> (para maiores detalhes sobre a hierarquia de generalização, ver seção 3.2.)

<def.agrupamento> ::=
grouping of <nome classe>

Declara que a classe em definição é um agrupamento de <nome classe> (maiores informações sobre a hierarquia de agrupamento podem ser encontradas na seção 3.2.).

<def.agregação> ::=
aggregation [redefinition] of <lista de classes>

Declara que as instâncias da classe em definição são compostas pela agregação de instâncias das classes listadas em <lista de classes>. O parâmetro **redefinition** especifica se <lista de classes> é uma redefinição da estrututra de agregação herdada das classes superiores, na hierarquia.

parameters : **with time** [**and lifetime** <constante> :
<unidade tempo>]

O parâmetro **with time** relaciona cada instância da classe a uma classe TIME_INTERVALL. O intervalo começa com o tempo de inserção e termina com o tempo atual ou, para instâncias removidas, com o tempo de remoção. Se **lifetime** é usado, instâncias podem pertencer à classe somente para o período de tempo especificado (e.g. 3:month).

on <evento> **when** <condição> **do** <ação>

Define uma <ação> a ser realizada quando ocorrer o evento <evento> e a <condição> for satisfeita.

<cláusula let> ::= **let** <variável> **be** (<termo>|<mensagem>)

A cláusula "let" atribui o valor de <termo> a <variável>.

No caso de mensagem, o valor retornado pela mensagem é atribuído a <variável>.

<mensagem> ::= "[" <objeto> <seletor> ["(" <parâmetros> ")"] "]"

onde

<objeto> é a instância ou classe receptora da mensagem;

<seletor> é o nome de um método definido na classe do objeto receptor;

<parâmetros> são os parâmetros reais passados para o método selecionado.

Exemplo: **let** minhaCaixa **be** [Caixa createInstance]

Esta declaração atribui o identificador do objeto criado na classe Caixa à variável minhaCaixa.

A estrutura geral de um método, no TOM, obedece à seguinte sintaxe:

```
<nome-método> ["(" <lista de parâmetros> ")"]
  [
   pre-conditions
        <predicado>
        [otherwise (warning ¦ cancel ¦ error)]
  ]
  [
   body
        <declarações>
  ]
  [
   post-conditions
        <predicado>
        [otherwise (warning ¦ cancel ¦ error)]
  ]
```

onde

<lista de parâmetros> é a lista de parâmetros de entrada;

<predicado> é uma conjunção de predicados primitivos que precisam ser satisfeitos a fim de que o corpo do método (declarações) possa ser executado;

<declarações> é uma sequência de operações primitivas, métodos e estruturas de controle do TOM.

Se alguma das pré-condições falha, a cláusula "otherwise" especifica qual a ação a tomar. Há três opções de ação para o caso de falha das pré-condições:

"warning" - a falha não evita a execução do método. O sistema apenas emite uma mensagem de advertência;

"cancel" - o método não é executado;

"error" - toda a cadeia de mensagens que originou a operação atual é cancelada.

5.3. Classes Básicas

A seguir é introduzido um esboço do modelo orientado a objetos TOM, como um sistema de metaclasses. No metanível são definidos os conceitos de classe, relacionamentos-classe, relacionamentos-instância, métodos-classe, métodos-instância, etc. O comportamento do modelo (métodos) não é descrito de forma integral, apenas sua sintaxe.

Muitas vezes será preciso especificar que um ou mais parâmetros são opcionais. A forma natural de representar este comportamento é através da hierarquia de generalização. Com o objetivo de facilitar a compreensão, adotamos uma simplificação expressa pelo símbolo {}, que significa um "string" vazio. Por exemplo, a expressão ('disjunctive'|{}) significa que o parâmetro "disjunctive" é opcional. A barra vertical significa "ou". A expressão (nome) significa um objeto identificado por "nome".

```
Metaclass TOM_CLASS
   class methods

      create (class_name)
         pre conditions
/* A1 */     not class_name in CLASS_NAME /* classes  distintas
                                              devem  ter
                                              nomes distintos */

            not (class_name) in self

         pos_conditions
            class_name in CLASS_NAME
            (class_name) in TOM_CLASS

      select(predicado:PREDICADO)

   instance relationships
      class_name : CLASS_NAME (1,1)
```

```
        class_relationships : CLASS_RELATIONSHIPS (0,*)
        class_methods : CLASS_METHODS (0,*)
        instance_relationships : INSTANCE_RELATIONSHIPS (1,*)
        instance_methods : INSTANCE_METHODS (0,*)
        identification : KEY_OR_TYPE (1,1)
        generalizations : ROLE_DEF (0,*)
        grouping : GROUPING_DEF (0,1)
        aggregation : AGGREGATION_DEF (0,1)
        parameters : TIME (0,1)
        has_objects : (class_name) (0,*)
        events : EVENT (0,*)

    instance methods
        update(nome_relaciomanento:string,novo_valor:DOM)

        delete()
            pre conditions
                has_objects(self) = {}

            pos conditions
                not "self" in TOM_CLASS

        createInstance

    identification
        keys are class_name

    parameters
        with time


Metaclass CLASS_NAME
    type string


Metaclass N
    type integer

Metaclass RELATIONSHIP
    instance relationships
        card : integer
        inv : RELATIONSHIP (1,1)

    keys are inherited
    with role tipo_relacionamento
        gives subclasses CLASS_RELATIONSHIPS, INSTANCE_RELATIONSHIPS
        parameters : covering, disjunctive

    aggregation of CLASS_NAME_A, NOME_REL, ':', CLASS_NAME_B,
                                                CARDINALIDADE

/* Originalmente, uma classe agregada é definida como uma
agregação de classes componentes. Para sermos capazes de
expressar a sintaxe do modelo, permitimos que uma constante (e.g.
':') faça parte de uma agregação. */
```

```
    class methods
       create (novorel : RELATIONSHIP)
          pre conditions

/* A2 */     let IR be [RELATIONSHIP.CLASS_NAME_A =
                                   novorel.CLASS_NAME_A].NOME_REL
/* Uma classe
   não pode
   ter dois              not novorel.NOME_REL in IR
   relacionamentos
   com o mesmo
   nome */

/* A3 */     if novorel.CARD_MAX <> '*' then
                    novorel.CARD_MIN <= novorel.CARD_MAX

/* A4 */     card(novorel) >= novorel.CARD_MIN
             if novorel.CARD_MAX <> '*' then
                 card(novorel) <= novorel.CARD_MAX

          pos conditions
             novorel in RELATIONSHIP
             novorel.NOME_REL in IR
 /* A6 */    inv(novorel) in RELATIONSHIP


Metaclass CARDINALIDADE
   aggregation of '(', CARD_MIN, ',', CARD_MAX, ')'

Metaclass NOME_REL
   type string

Metaclass CARD_MIN
   type integer

Metaclass CARD_MAX
   type integer

Metaclass CLASS_RELATIONSHIPS
   with role tipo_relacionamento_classe
      gives subclasses PRE_CLASS, POS_CLASS


Metaclass PRE_CLASS
   aggregation  redefinition of CLASS_NAME_A,  'pre_class',  ':',
                                CLASS_NAME_B, ('exclusive'|{})

Metaclass POS_CLASS
   aggregation redefinition of CLASS_NAME_A,  'post_class',  ':',
                               CLASS_NAME_B, ('exclusive'|{})


Metaclass INSTANCE_RELATIONSHIPS
```

```
    with role tipo_relacionamento_instancia
       gives subclasses HISTORY_INSTANCE_RELATIONSHIP,
                        ROLLBACK_INSTANCE_RELATIONSHIP


Metaclass RELATIONSHIP_TIME
   aggregation of 'with',  N,  ('history'|'rollback'|'temporal'),
                                                        'values'

Metaclass METHOD
   with role tipo_uso
      gives subclasses INSTANCE_METHODS, CLASS_METHODS
      parameters covering, disjunctive

   aggregation of METHOD_SIGNATURE, METHOD_IMPLEMENTATION

Metaclass METHOD_SIGNATURE
   aggregation of METHOD_NAME, '(', PARAMETERS, ')'

Metaclass METHOD_NAME
   type string

Metaclass PARAMETER_NAME
   type string

Metaclass PARAMETERS
   grouping of PARAMETER

Metaclass PARAMETER
   aggregation of PARAMETER_NAME, ':', DOM

Metaclass DOM
   with role tipo
      gives subclasses string,  integer,  real,  boolean, double,
                       char, TOM_Class
      parameters disjunctive


Metaclass METHOD_IMPLEMENTATION
   aggregation of METHOD_NAME, '(', PARAMETERS, ')',
                  METHOD_BODY

Metaclass METHOD_BODY
   aggregation of PRE_CONDITIONS, BODY, POS_CONDITIONS

Metaclass PRE_CONDITIONS
   grouping of CONDITION


Metaclass CONDITION
    aggregation of PREDICADO,(('AT', TEMPO)|{}),
                        (('otherwise',('warning'|'cancel'|'error'),
                         (MENSAGEM|{}))|{})
                       | ('let', VARIAVEL, 'be', TERMO)
```

```
Metaclass MENSAGEM
   type string

Metaclass PREDICADO
   with role tipo_predicado
      gives subclasses PREDICADO_PRIMITIVO,
                       COMPARACAO_RELACIONAL,
                       NEGACAO_PREDICADO,
                       PREDICADO_CONDICIONAL,
                       DISJUNCAO_PREDICADO,
                       PREDICADO_RELACIONAMENTO
      parameters : disjunctive, covering

Metaclass PREDICADO_PRIMITIVO
   with role tipo_predicado_primitivo
      gives  subclasses IN,  IS_REL,  IS_A,  ROLE_COMP,  IS_PART,
                        IS_ELEM

Metaclass IN
   aggregation of TERMO, 'in', CLASS_NAME

Metaclass IS_REL
   aggregation   of   'is_rel   (',NOME_ENTIDADE_SIMPLES,    ',',
                                  NOME_REL,                  ',',
                                  NOME_ENTIDADE_SIMPLES, ')'

Metaclass IS_A
   aggregation of 'is_a (', CLASS_NAME, ',', CLASS_NAME, ')'


Metaclass IS_PART
   aggregation of 'is_part (', NOME_ENTIDADE_SIMPLES, ',',
                               NOME_ENTIDADE_AGREGADA,  ')'

Metaclass IS_ELEM
   aggregation   of   'is_elem(',   NOME_ENTIDADE_SIMPLES,   ',',
                                    NOME_ENTIDADE_GRUPO, ')'

Metaclass COMPARACAO_RELACIONAL
   aggregation of TERMO, OPERADOR_RELACIONAL, TERMO

Metaclass OPERADOR_RELACIONAL
   with role tipo_operador_relacional
      gives subclasses '=', '<>', '<', '>', <=', '>='

Metaclass FUNCAO
   aggregation of NOME_REL, '(', TERMOS,  ')'

Metaclass TERMOS
   grouping of TERMO

Metaclass TERMO
   instance_relationships
       has_type : TOM_Class (1,1)
```

```
    with role tipo_termo
       gives    subclasses    CONSTANTE,    VARIAVEL,    FUNCAO,
                                        EXPRESSAO_ARITMETICA
       parameters : disjunctive


Metaclass NEGACAO_PREDICADO
   aggregation of 'not', PREDICADO

Metaclass PREDICADO_CONDICIONAL
   aggregation of 'if', PREDICADO, 'then', PREDICADO


Metaclass DISJUNCAO_PREDICADO
   aggregation of PREDICADO, 'or', PREDICADO

Metaclass PREDICADO_RELACIONAMENTO
   aggregation of NOME_REL, '(', TERMO, ',', TERMO, ')'

Metaclass NOME_ENTIDADE_SIMPLES
   type string

Metaclass NOME_ENTIDADE_AGREGADA
   type string


Metaclass BODY
   with role tipo_declaracao
      gives subclasses DECLARACOES_CONJUNTIVAS, DECLARACAO_DISJUNTIVA,
                       DECLARACAO_ITERATIVA

Metaclass DECLARACOES_CONJUNTIVAS
   grouping of DECLARACAO_CONJUNTIVA

Metaclass DECLARACAO_CONJUNTIVA
   aggregation of DECLARACAO, (('only if', PREDICADO)|{})

Metaclass DECLARACAO_DISJUNTIVA
   aggregation of 'case', OPCOES_CASE

Metaclass OPCOES_CASE
   grouping of OPCAO_CASE


Metaclass OPCAO_CASE
   aggregation of PREDICADO, ':', DECLARACAO


Metaclass DECLARACAO_ITERATIVA
   aggregation of 'for each', CONDICAO, 'do', DECLARACAO

Metaclass CONDICAO
   aggregation of PRE_CONDICAO,(('such that', PREDICADO)|{})
```

```
Metaclass PRE_CONDICAO
   keys are inherited
   with role tipo_pre_cond
      gives subclasses COND_IN, COND_CLASS parameters : covering

Metaclass COND_IN
   aggregation of VAR_ENTIDADE, 'in', (NOME_ENTIDADE_GRUPO|
                                       CLASS_NAME)

Metaclass COND_CLASS
   aggregation of 'class', VAR_CLASS


Metaclass VAR_ENTIDADE
   with role variavel
      gives subclasses 'x', 'y', 'z'

Metaclass VAR_CLASS
   with role variavel
      gives subclasses 'X', 'Y', 'Z'

Metaclass NOME_ENTIDADE_GRUPO
   type string

Metaclass DECLARACAO
   keys are inherited
   with role tipo_declaracao
      gives subclasses OPERACAO_PRIMITIVA, OPERACAO_CALL, CLAUSULA_LET
            parameters covering, disjunctive

Metaclass OPERACAO_PRIMITIVA
   aggregation of OPER_PRIMITIVA, (('at', TEMPO)|{})

Metaclass OPER_PRIMITIVA
   with role tipo_oper_primitiva
      gives  subclasses  CREATE,  DELETE,  GR_INSERT,  GR_DELETE,
                         ESTABLISH, REMOVE, MOVE, UPDATE

Metaclass OPERACAO_CALL
   aggregation of NOME_OPERACAO, '(', INPUT_PARAMETERS, ';',
                                      OUTPUT_PARAMETERS, ')'


Metaclass INPUT_PARAMETERS
   grouping of PARAMETER

Metaclass OUTPUT_PARAMETERS
   grouping of PARAMETER

Metaclass NOME_OPERACAO
   type string

Metaclass CLAUSULA_LET
   aggregation of 'let', VARIAVEL, 'be', TERMO
```

```
Metaclass CREATE
   aggregation of CLASS_NAME, 'create', '(', TERMO, ')'

Metaclass DELETE
   aggregation of TERMO, 'delete', (('(', CLASS_NAME, ')');{})

Metaclass GR_INSERT
   aggregation of NOME_ENTIDADE_GRUPO, 'gr-insert', '(', TERMO, ')'

Metaclass GR_DELETE
   aggregation of TERMO, 'gr-delete', (('(', NOME_ENTIDADE_GRUPO,
                                                        ')');{})


Metaclass ESTABLISH
   aggregation  of  NOME_REL, 'establish', '(' (TERMO1¦CLASS_NAME),
                                                   TERMO2, ')'

   class methods
      create (novorel : ESTABLISH)
         pre condition
            <has_type(novorel.TERMO1),novorel.NOME_REL,
/* A5 */       has_type(novorel.TERMO2)> in RELATIONSHIP

/*  Dois  objetos  estão  relacionados  somente  se  as   classes
correspondentes estão relacionadas */

         pos conditions
            novorel in ESTABLISH

            let  INVREL  be  [RELATIONSHIP
                 inv(<has_type(novorel.TERMO1),novorel.NOME_REL,
                      has_type(novorel.TERMO2)>)].NOME_REL

            <novorel.TERMO2,INVREL,novorel.TERMO1> in ESTABLISH


Metaclass REMOVE
   aggregation of '<', (TERMO1¦CLASS_NAME), TERMO2, '>', 'remove'

Metaclass MOVE
   aggregation of TERMO,  'move',  '(', CLASS_NAME1, CLASS_NAME2, ')'

Metaclass UPDATE
   aggregation of (TERMO¦CLASS_NAME),  'update',
                                  '(', NOME_REL, '=', TERMO, ')'

Metaclass POS_CONDITIONS
   grouping of CONDITION


Metaclass TEMPO
   keys are inherited
   with role tipo_especificacao_tempo
      gives subclasses EXPRESSAO_TEMPO, ESPECIFICACAO_UNIDADE,
```

```
                    TUPLA_TEMPO, INTERVALO_TEMPO

Metaclass EXPRESSAO_TEMPO
   aggregation   of   'clock.',   UNIDADE,   OPERADOR_ARITMETICO,
                                                VALOR_NUMERICO

Metaclass ESPECIFICACAO_UNIDADE
   aggregation of UNIDADE, ':', VALOR_NUMERICO

Metaclass TUPLA_TEMPO
   aggregation of ANO, MES, DIA, HORA, MINUTO, SEGUNDO

Metaclass INTERVALO_TEMPO
   aggregation of '<', TEMPO, 'to', TEMPO, '>'

Metaclass UNIDADE
   with role unidade_tempo
      gives subclasses 'day', 'month', 'year', 'hour', 'minute',
                       'second'

Metaclass OPERADOR_ARITMETICO
   with role tipo_operador
      gives subclasses '+', '-', '*', '/'

Metaclass KEY_OR_TYPE
   with role chave_ou_tipo
      gives subclasses KEY_DEF, TYPE_DEF

Metaclass KEY_DEF
   aggregation of 'Keys are', (LISTA_CHAVES ! 'inherited')

Metaclass LISTA_CHAVES
   grouping of CHAVE

Metaclass CHAVE
   type string

Metaclass TYPE_DEF
   aggregation of 'type', DOM, (('format', ESPECIFICACAO)!{})

Metaclass ROLE_DEF
   aggregation   of   'with   role',   NOME_PAPEL,   'gives   subclasses',
                      CLASSES_PREDICATE_OR_INDEX,
                (('parameters:',  ('disjunctive,'!{}),
                                  ('covering'!{}))!{})

Metaclass LIST_CLASSES
   aggregation of LISTA_CLASSES, 'explicit'

Metaclass NOME_PAPEL
   type string

Metaclass LISTA_CLASSES
   grouping of CLASS_NAME
```

```
Metaclass CLASSES_PREDICATE_OR_INDEX
   with role classe_predicado_ou_indice
      gives subclasses LIST_CLASSES, CONDICAO_PREDICADO,
                       CONDICAO_INDICE

Metaclass CONDICAO_PREDICADO
   aggregation of 'by predicate', PREDICADO

Metaclass CONDICAO_INDICE
   aggregation of 'using', NOME_REL, 'as index'

Metaclass GROUPING_DEF
   keys are inherited
   aggregation of 'grouping of', CLASS_NAME,
                    ('all'| 'explicit'| PREDICATE_OR_USING),
          (('parameters:', ('disjunctive,'|{}),
                           ('covering,'|{}), ('ordered'|{}))|{})

Metaclass PREDICATE_OR_USING
   with role predicado_ou_def_rel
      gives subclasses CONDICAO_PREDICADO, DEF_USING


Metaclass DEF_USING
   aggregation of 'using', NOME_REL

Metaclass AGGREGATION_DEF
   aggregation of 'aggregation', ('redefinition'|{}), 'of',
                    DEF_CLASSES, ('exclusive'|{})

Metaclass DEF_CLASSES
   with role tipo_agregacao
      gives subclasses AGREGACOES_BINARIAS, AGREGACAO_MULTIPLA

Metaclass AGREGACOES_BINARIAS
   grouping of AGREGACAO_BINARIA

Metaclass AGREGACAO_BINARIA
   aggregation of CLASS_NAME, CLASS_NAME, 'by', NOME_REL

Metaclass AGREGACAO_MULTIPLA
   aggregation of LISTA_CLASSES, ('all'| 'explicit')

Metaclass TIME
   keys are inherited
   aggregation of 'parameters: with time', (LIFETIME|{})


Metaclass LIFETIME
   aggregation of 'and lifetime', CONSTANTE, ':',
                                  ESPECIFICACAO_UNIDADE

Metaclass VALOR_NUMERICO
```

```
    type integer


Metaclass VARIAVEL
   aggregation of LETRA, LETRAS_OU_DIGITOS

Metaclass LETRA
   with role alfabeto
      gives subclasses 'A', 'B', 'C', ..., 'Z'

Metaclass LETRAS_OU_DIGITOS
   grouping of LETRA_OU_DIGITO parameters: ordered

Metaclass LETRA_OU_DIGITO
   with role letra_ou_digito
      gives subclasses LETRA, DIGITO

Metaclass DIGITO
   with role naturais
      gives subclasses '0', '1', '2', ..., '9'

Metaclass CONSTANTE
   with role tipo_constante
      gives subclasses integer, real, string
      parameters: disjunctive

Metaclass EXPRESSAO_ARITMETICA
   with role tipo_expressao_aritmetica
      gives subclasses EXP_BINARIA, EXP_UNARIA

Metaclass EXP_BINARIA
   aggregation of TERMO, OPERADOR_ARITMETICO, TERMO

Metaclass EXP_UNARIA
   aggregation of '-', TERMO

Metaclass EVENTO
   aggregation of 'event', NOME_EVENTO, '(', PARAMETERS, ')',
                  'on', OCORRENCIA, 'condition', PREDICADO,
                  'trigger', NOME_TRIGGER, PRE_CONDITIONS,
                             BODY, POS_CONDITIONS

Metaclass NOME_EVENTO
   type string

Metaclass NOME_TRIGGER
   type string

Metaclass OCORRENCIA
   aggregation of 'change (', EXPRESSAO_TEMPO, ')'

Metaclass EVENT
   aggregation of 'on', NOME_EVENTO, 'when', PREDICADO, 'do',
                                                 BODY
```

5.4. Tempo e Versionamento

Podemos classificar os bancos de dados de acordo com as suas habilidades para o processamento do tempo. Este critério de classificação, segundo Snodgrass [SA87], dá origem a quatro tipos de Bancos de Dados(BDs):

1- BDs Instantâneos - são os BDs clássicos que mantêm apenas a informação sobre os dados válidos atualmente.

2- BDs Históricos - associam a cada unidade de dados no BD (tupla, entidade, objeto ou atributo) uma informação sobre o tempo de existência daquele dado.

3- BDs de "Rollback" - mantêm o tempo de transação, os pontos de tempo em que a informação torna-se conhecida pelo BD (início) e no qual a informação deixa de ser válida (fim).

4- BDs Temporais - combinam as habilidades de processamento de tempo histórico e de transação (rollback).

Há três níveis de "timing" para sistemas de bancos de dados. O nível mais alto é o **"Timing" de Esquema**, que reflete o histórico do esquema de aplicação, das classes e suas estruturas. O **"Timing" de Objeto** lida com objetos temporais e o **"Timing" de Propriedade** mantêm valores anteriores de um relacionamento.

O tempo de vida de um elemento, dado por um ou mais intervalos de tempo, é representado pelos pontos de tempo "begin" (início) e "end" (fim) desses intervalos. O banco de dados atual é composto dos elementos cujo ponto-final está no futuro ou é

igual à função "now" (agora). No caso de tempo de "rollback", o ponto-final do intervalo é usado somente se o fato registrado no ponto-inicial se torna falso [AHN86].

"TIMING" DE OBJETO

O "Timing" de Objeto é regido por duas metaclasses, HistoryObject e RollbackObject, com os relacionamentos e métodos temporais correspondentes.

```
Metaclass HistoryObject
   instance relationships
      historyBegin : Timepoint
      historyEnd   : Timepoint

   events INSTEAD C create (e)
               DO let t1 be now
                  C create (e,t1,now)

          INSTEAD C create (e,t)
               DO C create (e,t,now)

          INSTEAD e delete()
               DO let t2 be now
                  e  update (historyEnd = t2) /* move o objeto
                                                 p/ o passado */

          INSTEAD e delete(t)
               DO e update (historyEnd = t)

Metaclass RollbackObject
   instance relationships
      rollbackBegin : Timepoint
      rollbackEnd   : Timepoint

   events INSTEAD C create (e)
               DO let t1 be now
                  C create (e,t1,now)

          INSTEAD e delete()
               DO let t2 be now
                  e  update (rollbackEnd = t2) /* o objeto  deixa
                                                  de ser válido  */
```

Uma classe de aplicação com tempo histórico e/ou de

"rollback" deve ser declarada como subclasse de HistoryObject e/ou RollbackObject, a fim de herdar os relacionamentos e eventos adicionais.

As duas primeiras propriedades de HistoryObject, historyBegin e historyEnd, fornecem os intervalos de tempo durante os quais o objeto foi considerado como uma instância da classe. Se o valor de historyEnd for igual à função "now", o objeto é uma instância atual da classe. Se o histórico contém mais de um intervalo de tempo, isto significa que o objeto foi uma instância da classe por várias vezes. Neste caso, um dos intervalos (não necessariamente o último) pode ter historyEnd = now, com o mesmo significado de um único intervalo.

As propriedades rollbackBegin e rollbackEnd especificam um intervalo durante o qual o objeto esteve corretamente declarado como instância da classe. Na maioria dos casos, rollbackBegin coincide com historyBegin e rollbackEnd é sempre nulo. Isto significa que o tempo de transação coincide com o tempo lógico e a informação armazenada é reconhecidamente correta. Se rollbackBegin difere de historyBegin, o objeto se tornou conhecido para o BD (tempo de transação) num instante diferente daquele em que o objeto apareceu como instância da classe na realidade (tempo lógico). Um valor de Timepoint atribuído a rollbackEnd significa que, naquele instante, descobriu-se que o objeto estava erroneamente declarado como instância da classe, para o intervalo de tempo histórico correspondente.

"TIMING" DE PROPRIEDADE

O "Timing" de Propriedade de um relacionamento "rel" de uma classe "C" para uma classe de domínio "D" é regido por duas metaclasses, HISTORY_INSTANCE_RELATIONSHIP e ROLLBACK_INSTANCE_RELATIONSHIP, com os relacionamentos e eventos temporais correspondentes.

```
Metaclass HISTORY_INSTANCE_RELATIONSHIP
   instance_relationships
      has_old_values : RELATIONSHIP_TIME (1,1)
      historyBegin : Timepoint
      historyEnd   : Timepoint

   events
               INSTEAD rel establish (<e1,e2>)
                    DO let t1 be now
                       rel establish (<e1,e2>,t1,now)

               INSTEAD <e1,e2,t1,now> remove()
                    DO let t2 be now
                       <e1,e2> update (historyEnd = t2)


Metaclass ROLLBACK_INSTANCE_RELATIONSHIP
   instance_relationships
      has_old_values : RELATIONSHIP_TIME (1,1)
      rollbackBegin : Timepoint
      rollbackEnd   : Timepoint

   events
               INSTEAD rel establish (<e1,e2>)
                    DO let t1 be now
                       rel establish (<e1,e2>, t1, now)

               INSTEAD <e1,e2,t1,now> remove()
                    DO let t2 be now
                       <e1,e2> update (rollbackEnd = t2)
```

De acordo com esses eventos, sempre que um relacionamento específico é removido, o valor removido expresso por <e1,e2> é estabelecido como válido para o intervalo de tempo <t1,t2> onde t2 é o valor atual da função "now".

Um relacionamento com valores anteriores precisa ser

declarado como histórico, rollback ou temporal. Por exemplo

```
class C
   instance relationships
      rel : D (1,1) with 3 history values
```

declara que o relacionamento **rel** mantém informação sobre o tempo histórico dos três últimos valores do relacionamento. Se o tempo de "rollback" também é necessário, declaramos

```
class C
   instance relationship
      rel : D (1,1) with 3 temporal values
```

"TIMING" DE ESQUEMA

O "Timing" de Esquema é obtido aplicando-se os conceitos de "timing" de objeto no metanível, já que todas as classes são objetos nesse metanível. O "Timing" de Esquema é modelado pela definição de uma classe como instância de TemporalSchema. TemporalSchema é uma metaclasse definida com a seguinte estrutura:

```
Metaclass TemporalSchema
   specialization of TOM_CLASS
   instance relationships
      class_name : CLASS_NAME (1,1) with temporal values
      class_relationships   :   CLASS_RELATIONSHIPS  (0,*)   with
                                                  temporal values
      class_methods : CLASS_METHODS (0,*) with temporal values
      instance_relationships :  INSTANCE_RELATIONSHIPS (1,*) with
                                                  temporal values
      instance_methods   :  INSTANCE_METHODS (0,*)  with  temporal
                                                     values
      identification : KEY_OR_TYPE (1,1) with temporal values
      generalizations : ROLE_DEF (0,*) with temporal values
      grouping : GROUPING_DEF (0,1) with temporal values
      aggregation : AGGREGATION_DEF (0,1) with temporal values
      parameters : TIME (0,1) with temporal values
```

A estrutura de TOM_CLASS, redefinem-se os relacionamentos-instância com "timing" de propriedade (cláusula "temporal"). Isto significa que será mantido o histórico das atualizações no esquema.

Por exemplo, uma classe **Documento** poderia ser declarada como instância de TemporalSchema, assim

```
class Documento
   instance_of TemporalSchema
                .
                .
                .
```

A classe **Documento** é criada como instância de TemporalSchema. Sua estrutura contém todos os relacionamentos-instância definidos em TemporalSchema. Qualquer alteração no esquema de **Documento** (e.g., adicionando ou removendo relacionamentos, métodos, etc) provoca uma atualização nos relacionamentos-instância do metanível. Como estes relacionamentos são definidos com "timing" de propriedade temporal, o histórico das atualizações no esquema é mantido.

VERSIONAMENTO

Para aplicações de CAD, CASE e outros, toda a evolução da construção dos objetos é de central importância, o que se reflete na necessidade de modelagem de versões de objetos no banco de dados.

Como exemplos de modelos de dados de versão podemos citar o modelo de dados Servidor de Versão [KATZ90], Haskin & Lorie, McLeod et alii, Dittrich & Lorie, Batory & Kim, entre

outros.

O modelo Servidor de Versão baseia-se em três relacionamentos ortogonais: históricos de versão, configurações e equivalências ("views"). O conceito de configurações não foi considerado no esquema de controle de versões do TOM. O modelo Servidor de Versão não é realmente um modelo orientado a objetos, estando mais associado aos modelos de dados semânticos, ao passo que , no TOM, os conceitos de versões e orientação a objetos são considerados de forma conjunta.

Integramos ao TOM os conceitos de versionamento desenvolvidos para o modelo de dados VODAK [SCHI89].

Os conceitos de versionamento fornecem um modelo para o processo de desenvolvimento de um objeto. Existem diversas formas de versionamento que podem ser classificadas em: versionamento temporizado ou numerado; e versionamento único ou múltiplo. No versionamento temporizado, "time stamps" são associados às versões, refletindo o tempo de existência de uma versão. No versionamento numerado, um número é associado a cada nova versão de um objeto. A ordem dos números determina a ordem das versões. No versionamento único, há uma sequência única de versões. E no versionamento múltiplo, várias cópias parciais ou totais de um objeto podem existir simultaneamente formando um dígrafo acíclico de versões.

Versões distintas de um objeto sempre ocorrem em momentos distintos e/ou diferem de alguma forma pelo seu conteúdo.

O mecanismo de versão do TOM baseia-se nos conceitos de OBJETOS-G e OBJETOS-V [WIL87]. OBJETO-G (objeto genérico) é um

conceito abstrato que descreve as propriedades comuns a todas as versões do objeto. Estas propriedades são especificadas numa CLASSE-G (G_CLASS). OBJETOS-V são as versões individuais de um OBJETO-G e ocorrem como instâncias de uma CLASSE-V (V_CLASS). Os objetos-v podem ser todos instâncias de uma mesma CLASSE-V ou de várias CLASSES-V. Os objetos-v estão relacionados entre si por relacionamentos especiais e formam um grafo acíclico direcionado, chamado **grafo de versão.** Além disso, eles estão relacionados a um único objeto-g, pela relação "versão-de".

Um objeto versionado tem uma evolução representada por um grafo de versão. Os arcos entre objetos-v representam o progresso do objeto nessa evolução.

Existem quatro tipos de arcos no grafo de versão, refletindo as relações de evolução de uma versão para outra:

1. relação seguinte - representa uma sequência cronológica única de uma versão para outra;

2. relação alternativa-de - significa que as várias versões seguintes são mutuamente exclusivas, refletindo tentativas alternativas de desenvolvimento do objeto;

3. relação variante-de - significa que todas as versões seguintes são variantes válidas do mesmo objeto;

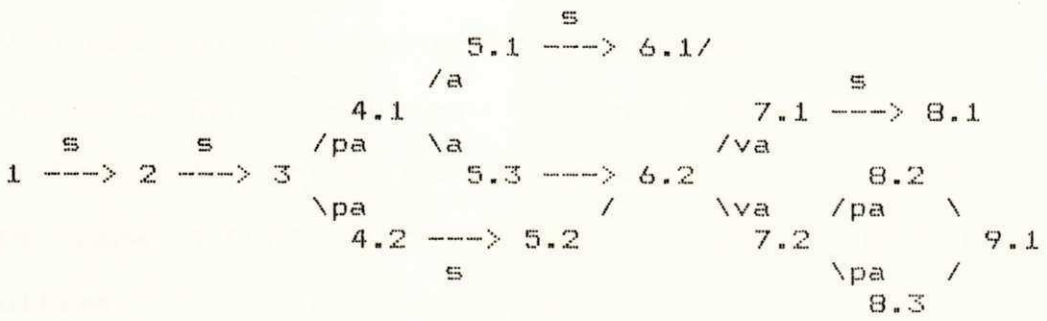
4. relação parte-de - significa que as versões seguintes são partes do mesmo objeto desenvolvidas em separado.

Ilustramos o uso de todas essas modalidades de versionamento por um exemplo de um escritório de projeto de

software.

EXEMPLO

Considere o projeto de um banco de dados para engenharia de software. A figura 5.2 mostra um possível grafo de versão do projeto de um software.



Notação : s=seguinte; a=alternativa; va=variante; pa=parte

Explicação : A análise foi feita em três versões subsequentes. Dois projetistas recebem o produto final da análise (3.) e iniciam o projeto de partes distintas do sistema. O projetista com a versão 4.1 faz uma tentativa de desenvolvimento via 5.1 e 6.1 mas não é bem sucedido. Ele retorna a 4.1 e, como uma outra alternativa, termina sua parte com 5.3. O outro fez sua parte com 5.2. Os dois resultados são integrados em 6.2, a versão final do projeto. O usuário precisa de uma aplicação em dois sistemas operacionais diferentes, por isso duas implementações variantes foram iniciadas como 7.1 e 7.2. O primeiro programador terminou seu trabalho com 8.1 e o outro dividiu a programação em duas partes (ou módulos), como 8.2 e 8.3 e integrou os módulos na versão final 9.1. Portanto, o produto de software final é composto de três visões: [Análise = versão 3.; Projeto = versão 6.2; Programação = versões 8.1 e 9.1].

Figura 5.2 - Exemplo de grafo de versão

O sistema tem duas metaclasses G_CLASS e V_CLASS. O usuário define sua classe de aplicação ProdutoSoftware como

subclasse de G_CLASS, a fim de herdar o relacionamento "has_versions" (tem_versões) definido lá.

```
class ProdutoSoftware
   specialization_of G_CLASS
   instance_relationships
      custo_total : CUSTO (1,1)
      responsavel : NOME (1,1)
```

As versões individuais ocorrem em três subclasses de V_CLASS: Análise, Projeto e Programação, que são declaradas visões de ProdutoSoftware. Em Projeto é possível aplicar diversas versões alternativas e em Programação, várias partes e variantes. Em cada V_CLASS todas as versões são temporárias, exceto a última.

```
class Analise
   specialization_of V_CLASS
   view_of ProdutoSoftware
   instance_relationships
      nextVersion : <ANALISE,PROJETO> (1,1)
      previousVersion : ANALISE (1,1)

class Projeto
   specialization_of V_CLASS
   view_of ProdutoSoftware
   instance_relationships
      previousVersion : <ANALISE,PROJETO> (1,1)
      nextVersion : <PROJETO,PROGRAMACAO> (1,1)
      alternative : PROJETO (2,*)

class Programacao
   specialization_of V_CLASS
   view_of ProdutoSoftware
   instance_relationships
      nextVersion : PROGRAMACAO (1,1)
      previousVersion : <PROJETO,PROGRAMACAO> (1,1)
      part : PROGRAMACAO (2,*)
      variant : PROJETO (2,*)
   with role ambiente_operacional
      gives subclasses PROGRAMACAO_DOS, PROGRAMACAO_UNIX
```

As metaclasses do sistema de versionamento, G_CLASS e

V_CLASS, são descritas como instâncias de TOM_CLASS, assim:

```
metaclass G_CLASS
   instance relationships
      has_versions : V_CLASS (1,*) inverse_of g_object
      identification : O_ID (1,1)

   class methods

      newVersionedObject
         pre conditions
            v in O_ID
            not (v) in self

         pos conditions
            (v) in self
```

A propriedade **has_versions** é herdada pelas classes de aplicação quando estas são declaradas como subclasse de G_CLASS. O relacionamento **identification** associa cada objeto genérico ao seu identificador.

```
metaclass V_CLASS
   specialization of HistoryObject
   instance relationships
      nextVersion : V_CLASS (1,*)
      previousVersion : V_CLASS (0,*)
      g_object : G_CLASS (1,1) inverse_of has_versions

   instance methods

      next                          /* mensagem enviada a um objeto-v */
         pos conditions
            let v be [V_CLASS createInstance]

/* createInstance é um método definido em TOM_CLASS */

            historyEnd(self) = historyBegin(v)
            g_object(self) = g_object(v)
            nextVersion(self) = v


      alternative_of           /* mensagem enviada a um objeto-v a
                                  ser substituído por nova  versão
                                  com a mesma previousVersion */
         pre conditions
            #(previousVersion(self)) = 1

         pos conditions
```

```
            let v be [previousVersion(self) next]
            historyBegin(v) >= historyEnd(self)


        variant_of      /* mensagem enviada a um objeto-v para criar
                           uma  variante v com estrutura Sv comum  a
                           Sself */
            pre conditions
               #(previousVersion(self)) = 1

            pos conditions
               let v0 be previousVersion(self)
               let v be [v0 next]
               Sv0 = intersec(Sv,Sself)
               historyEnd(self) = historyEnd(v)


        part_of_v
            pre conditions
               #(previousVersion(self)) = 1

            pos conditions
               let v0 be previousVersion(self)
               let v be [v0 next]
               contained(Sv0, union(Sself,Sv))
               intersec(Sself,Sv) = {}       /* exceto referência a */
                                             /* g_object                */
               historyEnd(self) = historyEnd(v)


        integrate(v)      /* integra o receptor da mensagem com v */
            pre conditions
               #(nextVersion(v)) = 0
               g_object(v) = g_object(self)
               historyEnd(v) = now
               historyEnd(self) = now

            pos conditions
               let vinteg be [self next]
               nextVersion(v) = vinteg
               contained(Svinteg, union(Sself,Sv))
               intersec(Sself,Sv) = {}      /* exceto referência a */
                                            /*                g_object */
```

O relacionamento **nextVersion** fornece a próxima versão, em relação à versão atual. O relacionamento **previousVersion** retorna a versão anterior à atual. E a propriedade **g_object** fornece o objeto genérico associado com uma dada versão. [SCHI89]

UFPb / BIBLIOTECA / PRAT

5.5. Classes Dinâmicas

A seguir, mostramos a modelagem dos axiomas dinâmicos e efeitos colaterais no meta-esquema. Cada método e evento apresenta a referência ao respectivo axioma ou efeito colateral modelado.

```
Metaclass CREATE
   aggregation of CLASS_NAME, 'create', '(', TERMO, ')'

   instance methods
      do (class_name, termo)
         pre conditions
            is_a (class_name, D, p)
/* AD3 */   p(class_name,termo)

         pos conditions
            termo in (class_name)

      do (class_name, termo)
         pre conditions
            covering (D,C1,...,Cn)
/* AD4 */   is_a (class_name, D, p)
            termo in D

         pos conditions
            termo in (class_name)

      do (class_name, termo)
         pre conditions
            timed(C',class_name,I)
/* AD5 */   not <termo, <t,now>> in C'

         pos conditions
            termo in (class_name)


   events
            ON (CLASS_NAME) create (TERMO)
/* EC1 */   WHEN is_a (CLASS_NAME,D)
                     not TERMO in D
            DO D create (TERMO)

            ON (CLASS_NAME) create (TERMO)
/* EC2 */   WHEN is_a (CLASS_NAME,D)
                     disjuntivo(D,C1,...,Cn)
                     TERMO in Ci
                     CLASS_NAME <> Ci
```

```
            DO TERMO delete

/* EC4 */   ON r establish (<TERMO,y>) OR
                  <TERMO,y> remove
            WHEN not TERMO in (CLASS_NAME)
                     is_a (CLASS_NAME,D,p)
            DO (CLASS_NAME) create (TERMO)

            ON <TERMO,y> remove OR
                   r establish (<TERMO,y>)
            WHEN TERMO in (CLASS_NAME)
            DO TERMO delete

/* EC5 */   ON D create (TERMO)
            WHEN is_a (CLASS_NAME,D)
                     pc(TERMO)
            DO (CLASS_NAME) create (TERMO)

/* EC9 */   ON rel establish (<TERMOi,TERMOj>)
            WHEN agregado_r(CLASS_NAME, C1, ..., Cn, R)
                     TERMOi in C1  . . .  TERMOn in Cn
                     rel in R
                     is_rel (Ci,rel,Cj)
                     not is_rel (TERMOi,rel,TERMOj)
            DO (CLASS_NAME) create (<TERMOi,...,TERMOn>)

/* EC10 */  ON D create (y)
            WHEN agregado(D,C1,...,Cn)
            DO if is_part(TERMOi,y)
                        then Ci create (TERMOi)

/* EC11 */  ON D create (y)
            WHEN agregado_r(D,C1,...,Cn,R)
                     is_rel(Ci,rel,Cj)
                     rel in R
            DO rel establish (<TERMOi,TERMOj>)
                   if is_part(TERMOi,y)
                      then Ci create (TERMOi)

/* EC13 */  ON CLASS_NAME create (TERMO)
            WHEN covering(CLASS_NAME,G,p)
                     not {TERMO} in G
            DO G create ({TERMO})
                   p(TERMO,{TERMO})

/* EC15 */  ON G create (g)
            WHEN is_elem(CLASS_NAME,G)
            DO if TERMO in g AND
                  not TERMO in CLASS_NAME
                        then CLASS_NAME create (TERMO)

/* EC20 */  ON CLASS_NAME create (TERMO)
            WHEN timed (CLASS_NAME',CLASS_NAME,I)
            DO CLASS_NAME' create (<TERMO,(t,now)>)
```

```
/* EC24 */ ON D create (TERMO)
           WHEN excl_pre(C,D)
           DO TERMO delete (C)

/* EC25 */ ON TERMO delete (C)
           WHEN excl_pos(C,D)
           DO D create (TERMO)


/* EC28 */ ON g createVersion (v)
           WHEN not g in G_CLASS
           DO G_CLASS create (g)
              g_object establish (<v,g>)


Metaclass DELETE
   aggregation of TERMO, 'delete', (('(', CLASS_NAME, ')')|{})

   instance_methods
      do (termo,class)
         pre condition
            termo in class

         pos condition
            not termo in class

   events

           ON TERMO delete (C)
/* EC3 */  WHEN is_a (C,D)
                    covering(D,C1,...,Cn)
                    Ci <> C
                    not TERMO in Ci
           DO TERMO delete (D)

/* EC6 */  ON TERMO delete (D)
           WHEN is_a (C,D)
                    TERMO in C
           DO TERMO delete (C)

/* EC7 */  ON TERMO delete
           WHEN is_part(C,D)
                    TERMO in C
                    agregado (y,TERMO,TERMO2,...,TERMOn)
                    y in D
           DO y delete

/* EC8 */  ON <TERMO,rel,TERMO2> remove
           WHEN agregado_r(y,TERMO,...,TERMOn,R)
                    y in D
                    TERMO in C
                    rel in R
           DO y delete

/* EC16 */ ON g delete (G)
```

```
            WHEN covering(C,G)
                    TERMO in g
                    not TERMO in h
                    h in G
                    h <> g
            DO TERMO delete (C)

/* EC18 */  ON TERMO delete
            WHEN is_rel(TERMO,rel,TERMO2)
                    TERMO in C
            DO <TERMO,rel,TERMO2> remove

/* EC21 */  ON TERMO delete (C)
            WHEN timed(C',C,I)
            DO if <TERMO,(t,now)> in C'
                  then <TERMO,(t,now)> delete
                       C' create (<TERMO,(t,t2)>)


/* EC29 */  ON g delete (G_CLASS)
            WHEN g = g_object(vi)
            DO vi delete (V_CLASS)



Metaclass ESTABLISH
   aggregation       of       NOME_REL,       'establish',       '(',
                                 (TERMO1|CLASS_NAME), TERMO2, ')'

   instance_methods
      do (nome_rel, termo1, termo2)
         pre_conditions
            <has_type(termo1),   nome_rel,  has_type(termo2)>  in
                                                          RELATIONSHIP
/* AD1 */    (NOME_REL.CARD_MAX = '*' OR
              let IR be [(NOME_REL).TERMO1 = termo1]
              card(IR) < NOME_REL.CARD_MAX)

         pos_conditions
            <termo1,nome_rel,termo2> in (NOME_REL)

   events

/* EC19 */  ON rel establish (<TERMO1,TERMO2>)
            WHEN not is_rel(TERMO2,inv(rel),TERMO1)
            DO inv(rel) establish (<TERMO2,TERMO1>)

/* EC22 */  ON rt establish (<TERMO1,TERMO2>)
            WHEN old(r')
                    t = now
            DO R' create (<TERMO1,TERMO2,(t,now)>)
```

```
Metaclass REMOVE
   aggregation  of '<',  (TERMO1|CLASS_NAME),  NOME_REL,  TERMO2,
                                    '>', 'remove'

   instance methods
      do (nome_rel, termo1, termo2)
         pre conditions
            let IR be [NOME_REL.TERMO1 = termo1]
/* AD2 */     card(IR) > NOME_REL.CARD_MIN

         pos conditions
            not <termo1,nome_rel,termo2> in (NOME_REL)



   events

/* EC23 */ ON <TERMO1,rt,TERMO2> remove
           WHEN old(r')
                   t = now
           DO if <TERMO1,TERMO2,(t0,now)> in R'
                 then <TERMO1,TERMO2,(t0,now)> delete
                      R' create (<TERMO1,TERMO2,(t0,t)>)


Metaclass UPDATE
   aggregation of (TERMO|CLASS_NAME), 'update', '(', NOME_REL, '=',
                                                      TERMO, ')'

   events

/* EC26 */ ON a update (r = q)
           WHEN agg_herança (A,A1,...,An,r)
                    is_part(ai,a)
           DO ai update (r = q)

/* EC27 */ ON ai update (r = qi)
           WHEN agg_herança_comp (A,A1,...,An,r,θ)
                    is_part(ai,a)
                    q = θ (qi)
           DO a update (r = q)

/* EC30 */ INSTEAD v1 update
           DO  let v2 be [v1 next]
               v2 update


Metaclass GR_INSERT
   aggregation of NOME_ENTIDADE_GRUPO,  'gr_insert',  '(', TERMO, ')'

   instance methods
      do (nome_ent_grupo, termo)
         pre conditions
            is_elem(C, nome_ent_grupo, p)
```

```
/* AD6 */    p(termo,g)

          pos_conditions
             termo in (nome_ent_grupo)

    events

/* EC12 */ ON C create (x)
           WHEN is_elem(C,G)
                     g in G
                     p(x,g)
           DO g gr_insert (x)

/* EC17 */ ON g gr_insert (TERMO)
           WHEN disjuntivo(C,G)
           DO if TERMO in h
                          h <> g
                    then TERMO gr_delete (h)


Metaclass GR_DELETE
    aggregation of TERMO, 'gr_delete', (('(', NOME_ENTIDADE_GRUPO,
                                                    ')'):{})
    events
           ON TERMO delete (C)
/* EC14 */ WHEN is_elem(C,G,p)
                     TERMO in g
           DO TERMO gr_delete (g)
```

6. Meta-Esquema do Modelo Relacional

Neste capítulo ilustramos a modelagem de um modelo qualquer, utilizando a linguagem do metanível. O modelo de dados a ser definido é o modelo relacional [DATE86].

O objetivo desta ilustração é apenas mostrar, brevemente, o processo de modelagem de um modelo. Por isso, não temos a pretensão de descrever profundamente o modelo relacional.

```
Metaclass RELATION_CLASS
   class_methods
      create(relationName, atributosR, chavesR)
         pre_conditions
            not relationName in RELATION_NAME
            not (relationName) in self

         pos_conditions
            attributes(relationName) = atributosR
            keys(relationName) = chavesR
            relationName in RELATION_NAME
            (relationName) in RELATION_CLASS

   instance_relationships
      relation_name : RELATION_NAME (1,1)
      attributes : ATTRIBUTE (1,*)
      keys : ATTRIBUTE (1,*)
      instances : TUPLE (1,*)

   instance_methods

      uniao (relation : RELATION_NAME)
         pre_condition
            attributes(self) = attributes(relation)

         pos_conditions
            let    R    be   [RELATION_CLASS    create    (nomeR,
                              attributes(relation), keys(relation))]

            instances(R) = instances(self) U instances(relation)
            R in RELATION_CLASS


      produto_cartesiano (relation : RELATION_NAME)
         pos_conditions
            let R be [RELATION_CLASS create (nomeR,
                        attributes(self) U  attributes(relation),
```

```
                    Keys(self) U keys(relation))]
        if r in self
               s in relation
            then <r,s> in R

        R in RELATION_CLASS


projecao (atributosP : ATTRIBUTE)
    pre condition
        atributosP in attributes(self)

    pos condition
        let R be [RELATION_CLASS create (nomeR,atributosP)]
        let instances(R) be [self projecao (atributosP)]
        R in RELATION_CLASS


selecao (condicao)
    pos conditions
        let  R be [RELATION_CLASS create (nomeR,  atributosR,
                                          chavesR) ]
        attributes(R) = attributes(self)
        if condicao
           then instances(R) = instances(self)
        R in RELATION_CLASS


diferenca (relation : RELATION_NAME)
    pre condition
        attributes(self) = attributes(relation)

    pos conditions
        let R be [RELATION_CLASS create  (nomeR,
                        attributes(relation), keys(relation))]
        instances(R) = instances(self) - instances(relation)
        R in RELATION_CLASS


intersecao (relation : RELATION_NAME)
    pre conditions
        attributes(self) = attributes(relation)

    pos conditions
        let R be [RELATION_CLASS create (nomeR,
                        attributes(relation), keys(relation))]
        instances(R) = instances(self) ∩ instances(relation)
        R in RELATION_CLASS


juncao (relation : RELATION_NAME, condicao)
    pos conditions
        let R1 be [self produto_cartesiano (relation)]
        let R2 be [R1 selecao (condicao)]
        R2 in RELATION_CLASS
```

```
        juncao_natural (relation : RELATION_NAME)
            pos conditions
                let R1 be [self produto_cartesiano (relation)]
                attributes(R1) = attributes(self) U attributes(relation)
                let    R2    be   [RELATION_CLASS   create    (nomeR,
                                        attributes(R1), keys(R1))]
                if  attributes(instances(self))   =
                        attributes(instances(relation))
                    then instances(R1) in instances(R2)
                R2 in RELATION_CLASS


    identification
        keys are relation_name


Metaclass RELATION_NAME
    type string

Metaclass ATTRIBUTE
    aggregation of ATTRIBUTE_NAME, DOMAIN


Metaclass ATTRIBUTE_NAME
    type string

Metaclass DOMAIN
    with role tipo_dominio
        gives subclasses CHAR, INTEGER, SMALLINT, FLOAT
```

7. Conclusões e Sugestões

No presente trabalho, descrevemos formalmente a sintaxe e a semântica do modelo de dados orientado a objetos TOM, usando um metanível onde é possível especificar qualquer modelo de dados. Para que um modelo de dados se torne realmente adequado à modelagem conceitual, ainda resta muito a fazer.

A maioria dos modelos de dados orientados a objetos consideram objetos como coisas que são identificáveis no banco de dados. Por essa definição, a caracterização de objetos se torna um detalhe de implementação, deixando de ser uma decisão conceitual. Na nossa opinião, esse não é o objetivo de um modelo de dados semântico. Nós demos uma definição mais liberal de objeto que é determinada por uma melhor orientação para o mundo real.

Além disso, bases de conhecimento precisam de um conceito mais genérico de **fato**. Por exemplo, o fato "João quebrou a perna ontem, e está no Hospital Sta. Clara" envolve os objetos João, sua perna, e Hospital Sta. Clara, o evento de quebrar uma perna, o relacionamento está_em e algumas informações temporais. Este fato é um objeto? Se sim, de que classe? Se não, qual é a fronteira entre objetos e fatos? Esta fronteira é conceitual ou interna? Como o fato é uma informação (implicitamente) armazenada no banco de dados, é mais correto falarmos de Base de Fatos ao invés de Banco de Dados ou de Objetos.

Um primeiro passo nessa direção é libertar os objetos da necessidade do identificador único. No TOM, a presença de instâncias de uma classe pode, em alguns casos, ser indicada por

um relacionamento-classe apenas, fornecendo o total de membros da classe. O próximo passo em direção à base de fatos é a aceitação de objetos livres sem que pertençam a uma classe.

Uma outra extensão necessária é a inclusão de informações incompletas tais como valores nulos, fatos negativos, valores indiretos, etc. Um mecanismo de herança para agrupamento precisa ainda ser explorado.

Referências Bibliográficas

[AHN86] AHN, I.. Towards an Implementation of Database Management Systems with Temporal Support. In IEEE Conference on Data Engineering, Los Angeles, 1986.

[BDZ89] BANCILHON, François; DITTRICH, Klaus; ZDONIK, Stanley; ATKINSON, Malcolm; DEWITT, David; MAIER, David. The Object-Oriented Database System Manifesto. In First International Conference on Deductive and Object-Oriented Databases, Kyoto, Japan, Dez, 1989, p.25-41.

[BDK89] BANCILHON, François; DELOBEL, Claude; KANNELAKIS, Paris. The O2 Book. GIP Altaïr, France, 1989.

[BN78] BILLER, H & NEUHOLD, E. J. Semantics of Data Bases: The Semantics of Data Models. Inf. Systems, vol. 3, 1978.

[ANSI86] BURNS, T. Reference Model for DBMS Standardization. ACM SIGMOD RECORD, 15(1):19-58, 1986.

[CHEN76] CHEN, P. P. The Entity Relationship Model: Towards a Unified View of Data. ACM - TODS, 1(1), 1976.

[CODD79] CODD, E. F. Extending the Database Relational Model to Capture More Meaning. ACM Trans. Database Systems, 4(4), Dez, 1979.

[COX86] COX, Brad J. Object-Oriented Programming - An Evolutionary Approach. Addison-Wesley, 1986.

[CH86] COX, Brad J. & HUNT, Bill. Objects, Icons and Software-ICs. Byte, 11(9), p.161-176, Ago, 1986.

[CUN88] CUNHA, Joseluce de Farias. Mapeamento do Modelo Semântico de Dados THM para o Modelo Relacional. Tese de Mestrado, Universidade Federal da Paraíba, 1988.

[DATE86] DATE, C. J.. Introdução a Sistemas de Bancos de Dados. Editora Campus, Rio de Janeiro, 1986, 4a. ed..

[DITT87] DITTRICH, Klaus R. Object-Oriented Database Systems: A Workshop Report. In 5th ER-Conference, Dijon, Nov 1986, North-Holland, 1987, p.51-66.

[FIL87] FILHO, Antenor Ferreira. O Sistema de Efeitos Colaterais em THM. Tese de Mestrado, Universidade Federal da Paraíba, 1987.

[GJ89] GARVEY, M. A. & JACKSON, M. S. Introduction to Object-Oriented Databases. Information and Software Technology, 31(10), p.521-528, Dez, 1989.

[HM81] HAMMER, M. & McLEOD, D. Database Description with SDM: a Semantic Database Model. ACM Trans. Database Systems, 6(3), Set, 1981.

[HZ88] HEILER, Sandra & ZDONIK, Stanley. Views, Data Abstraction, and Inheritance in the FUGUE Data Model. In 2nd International Workshop on Object Oriented Database Systems, Bad MÜnster , 1988, p.225-241.

[HK89] HUDSON, Scott E. & KING, Roger. Cactis: A Self-Adaptive, Concurrent Implementation of an Object-Oriented Database Management System. ACM Transactions on Database Systems. New York, 14(3), p.291-321, Set, 1989.

[KATZ90] KATZ, R. H. Toward a Unified Framework for Version Modeling in Engineering Databases. ACM Computing Surveys, 22(4), Dez, 1990, 375-408.

[KEM87] KEMPER, Alfons et alii. An Object-Oriented Database System for Engineering Applications. In Sigmod, 1987.

[KIM90] KIM, W. Defining Object Databases Anew. Datamation, 36(3), p.33-40, Fev, 1990.

[KM82] KING, R. & McLEOD, D. The Event Database Specification Model. In Second Int. Conf. Databases: Improving Usability and Responsiveness, Jerusalem, Israel, 1982.

[KLA90] KLAS, Wolfgang. A Metaclass System for Open Object-Oriented Data Models. Tese de Doutorado, Technischen Universität Wien, 1990.

[LRV89] LECLUSE, Christophe; RICHARD, Philippe; VELEZ, Fernando. O2, Un Modèle de Données Orienté-objet. Publicação Interna GIP Altaïr, France, 1989.

[MF90] MOTZ, Regina & FONSECA, Décio. Estudo Formal da Estrutura de um Modelo de Dados Orientado a Objetos. In 5o. Simpósio Brasileiro de Banco de Dados, Rio de Janeiro, 1990, p.113-127.

[MT89] MOZAFFARI, Mojtaba & TANAKA, Yuzuru. ODM: An Object Oriented Data Model. New Generation Computing, OHMSHA & Springer-Verlag, 1989, p.3-35.

[NP88] NAVATHE, Shamkant B. & PILLALAMARRI, Mohan K. OOER: Toward Making the E-R Approach Object-Oriented. In 7th ER Conference, Italy, 1988, p.55-76.

[PAS86] PASCOE, Geoffrey A. Elements of Object Oriented Programming. Byte, 11(9), p.139-144, Ago, 1986.

[RAO88] RAO, Kotcherlakota V. Bapa et alii. The Design of Dynamo: A General-purpose Information Processing Model with a Time Dimension. In 2nd International Workshop on Object Oriented Database Systems, Bad Münster, 1988, p.286-291.

[SAN88] SANTOS, José Laurindo Campos. DESCONT-THM : Um Sistema para o Desenvolvimento de Esquemas Conceituais de Dados em THM (Temporal Hierarchic Data Model) Baseado em Frames. Tese de Mestrado, Universidade Federal da Paraíba, 1988.

[SCHI82A] SCHIEL, Ulrich. The Temporal-Hierarchic Data Model, Universität Stuttgart Institut für Informatik, Stuttgart, 1982 (em inglês).

[SCHI82B] SCHIEL, Ulrich & MISTRIK, Ivan. Using Object-Oriented Analysis and Design for Integrated Systems. Universität Stuttgart Institut für Informatik, Stuttgart, 1982 (em inglês).

[SCHI83] SCHIEL, Ulrich. An Abstract Introduction to the Temporal-Hierarchic Data Model (THM). In 9th VLDB, Florence, 1983, p.322-330 (em inglês).

[SCHI89] SCHIEL, Ulrich. VODAK Version Model (VVM). Relatório Técnico, 1989, GMD/IPSI, Darmstadt, Deutschland.

[SCHI91] SCHIEL, Ulrich. An Open Environment for Objects with Time and Versioning. In EastEurOOPe'91, Bratislava, CSFR, 1991, p.116-125.

[SER88] SERNADAS, A. et alii. Abstract Object Types: A Temporal Perspective. In Collequium on Temporal Logic and Specifications. A. Pnneli(ed.), Springer-Verlag, 1988, p.1-31.

[SHIP81] SHIPMAN, D. The Functional Model and the Data Language DAPLEX. ACM Transactions on Database Systems, Mar, 1981.

[SA87] SNODGRASS, R. & AHN, I.. A Taxonomy of Time in Databases. In SIGMOD'87, Austin, 1987.

[WIL87] WILKES, W.. Der Versionsbegriff und seine Modellierung in CAD/CAM Datenbanken. Dissertação de Doutorado, FernUniversitet Hagen, 1987.

[ZR88] ZHAO, Liping & ROBERTS, S. A. An Object-Oriented Data Model for Database Modelling, Implementation and Access. The Computer Journal. 31(2), 1988, p.116-123.

Apêndice

SINTAXE DA METALINGUAGEM DO TOM

```
[meta]class <nome classe>
   [specialization of <nome metaclasse>]
   [instance of <nome metaclasse>]
   [
    class relationships
       (<nome  rel>  ":" <classe relacionada>
          "(" <card_min>  "," <card_max> ")" )*
       (pre_class ":" <classe relacionada> [exclusive])*
       (post_class ":" <classe relacionada> [exclusive])*
   ]
   [class methods
       <def.método>+      ]
   [
    instance relationships
       (
        <nome rel> ":" <classe relacionada>
          "(" <card_min> "," <card_max> ")"
        [with <n> (history¦rollback¦temporal) values]
       )+
   ]
   [instance methods
       <def.método>+      ]

   ( keys are ( <lista de chaves>+ ¦ inherited )
   ¦ type <tipo> [ format <especificação> ]

   [<def.papel>*]
   [<def.agrupamento>]
   [<def.agregação>]
   [parameters :  with time [and lifetime <constante> :
                                 <unidade tempo>]      ]
   [on <evento> when <condição> do <ação>]



<def.papel> ::=
          with role <nome papel>
               gives subclasses (<lista de classes> [explicit] ¦
                    by predicate <predicado> ¦
                    using <relacionamento> as index
               [  parameters: [disjunctive,] [covering]  ]

<def.agregação> ::=
          aggregation [redefinition] of
                    ( <lista de classes> (all¦explicit) ¦
                      <classe-1, classe-2> by
                      <relacionamento>  )          [exclusive]

<def.agrupamento> ::=
```

```
            grouping of  <nome classe> ( all ¦ explicit ¦
                                         by  predicate <predicado> ¦
                                         using <relacionamento>  )
            [ parameters : [disjunctive,] [covering,] [ordered] ]



<def.método> ::= <nome método> ["(" <lista parâmetros> ")"]
                      [
                       pre_conditions
                          ( <predicado>   [at   <tempo>]
                             [otherwise (warning¦cancel¦error)
                               [<mensagem>] ]
                          ¦ <cláusula let> )+
                      ]
                      [body
                          <declaração conjuntiva> ¦
                          <declaração disjuntiva> ¦
                          <declaração iterativa>
                      ]
                      [
                       pos_conditions
                          idem pre_conditions
                      ]

<declaração conjuntiva> ::=
           ( <declaração> [only if <predicado>] )+
<declaração disjuntiva> ::=
           case ( <predicado> : <declaração> )+
<declaração iterativa> ::=
           for each <condição> do <declaração>
<condição>  ::= ( (<var.entidade> in (<entidade grupo>¦<classe>))
                        ¦   class   <var.classe>  )
                                [such that <predicado>]
<declaração>   ::=  <operação  primitiva>  ¦  <operação  call>  ¦
                    <cláusula let>
<operação primitiva> ::= ( <create> ¦ <delete> ¦ <establish>
                              ¦ <remove> ¦ <move>   ¦ <update>
                              ¦  <gr_insert> ¦ <gr_delete> )
                                          [ at <tempo> ]
<operação call> ::= <nome operação> "(" <input parameters> ";"
                                                  <output parameters>
)"
<clásula let> ::= let <variável> be <termo>
<create> ::= <classe> create "(" <termo> ")"
<delete>  ::= <termo> delete [ "(" <classe> ")" ]
<establish>      ::=  <relacionamento>  establish
                       "(" (<termo1>¦<classe>) "," <termo2> ")"
<remove>  ::= "<" (<termo1>¦<classe>) ["," <relacionamento>]  ","
                        <termo2> ">" remove
<move> ::= <termo> move "(" <classe1> "," <classe2> ")"
<update>  ::=  (<termo>¦<class>) update "(" <relacionamento>  "="
                                          <termo> ")"
<gr_insert> ::= <entidade grupo> gr_insert "(" <termo> ")"
<gr_delete> ::= <termo> gr_delete [ "(" <entidade grupo> ")" ]
```

```
<tempo>  ::=  (clock.  <unidade>  <operador  aritmético>
               <valor numérico>
              ¦ <especificação unidade>
              ¦ <tupla de tempo>
              ¦ <intervalo de tempo>
<unidade>  ::=  "day"  ¦ "month" ¦ "year" ¦ "hour" ¦  "minute"  ¦
                "second"
<especificação unidade> ::= <unidade> ":" <valor>
<intervalo de tempo> ::=  "<" <tempo> to <tempo> ">"
<tupla de tempo> ::= <ano><mês><dia><hora><minuto><segundo>
<predicado> ::= <predicado primitivo>
              ¦ <termo1> <operador relacional> <termo2>
              ¦ not <predicado>
              ¦ if <predicado1> then <predicado2>
              ¦ <predicado1> or <predicado2>
              ¦ <relac. de membros> "(" <termo1> "," <termo2> ")"

<termo> ::= <constante> ¦ <variável ¦ <função>
          ¦ <termo1> <operador aritmético> <termo2>
          ¦ "-" <termo>
<função> ::= <relacionamento membros> "(" <termos> ")"
               ¦ <relacionamento classe> "(" ")"
<predicado primitivo> ::= <in> ¦ <is_rel> ¦ <is_a> ¦ <role_comp> ....
                        ¦ <is_part> ¦ <is_elem>
<in> ::= <termo> in <classe>
<is_part> ::= is_part "(" <entidade simples> ","
                          <entidade agregada> ")"
<nome operação> ::= <identificador>
<nome parâmetro> ::= <identificador>
<nome classe> ::= <identificador>
<lista chaves> ::= <identificador>+
<variável> ::= <identificador>
<constante> ::= <número> ¦ <string>
<entidade> ::= <var-entidade> ¦ <constante>
<var-entidade> ::= "x" ¦ "y" ¦ "z"
<var-classe> ::= "X" ¦ "Y" ¦ "Z"
<entidade grupo> ::= <identificador> ¦ <constante>
<lista param> ::= <identificador> ¦ <constante>
                ¦ <lista param> "," <identificador>
                ¦ <lista param> "," <constante>
<relacionamento>  ::=  <relacionamento membro>
                     ¦ <relacionamento classe>
<op. aritmético> ::= "+" ¦ "-" ¦ "*" ¦ "/"
<valor numérico> ::= <digito> <digito>*
<op. relacional> ::= "<" ¦ ">" ¦ "<=" ¦ ">=" ¦ "=" ¦ "<>"
<relacionamento membro> ::= <termo1> <rel.memb> <termo2>
<relacionamento classe> ::= <classe> <rel.classe> <termo>
<rel. membro> ::= <identificador>
<rel. classe> ::= <identificador>
<entidade agregada> ::= "<" <termo> ( "," <termo> )* ">"
<classe> ::= <identificador>
<classe agregada> ::= <identificador>
<classe componente> ::= <identificador>
<classe de grupo> ::= <identificador>
<classe elemento> ::= <identificador>
```

```
<classe generalizada> ::= <identificador>
<subclasse> ::= <identificador>
<identificador> ::= <letra> ( <letra> ¦ <digito> )*
<letra> ::= "A" ¦ "B" ¦ "C" ... ¦ "Z"
<digito> ::= "0" ¦ "1" ¦ "2" ... ¦ "9"
```

Notação:

expressão em negrito
ou entre aspas = símbolo terminal

```
<identificador>          = símbolo não terminal
[cláusula]             = cláusula opcional
(expressão)*           = expressão ocorre 0 ou mais vezes
(expressão)+           = expressão ocorre 1 ou mais vezes
¦                        = ou
```

ERRATA

Página 84

1. No lugar de

```
Metaclass DECLARACAO
   keys are inherited
   with role tipo_declaração
      gives subclasses OPERACAO_PRIMITIVA, OPERACAO_CALL,
                                           CLAUSULA_LET
            parameters : covering, disjunctive
```

leia-se

```
Metaclass DECLARACAO
   keys are inherited
   with role tipo_declaração
      gives subclasses OPERACAO_PRIMITIVA, OPERACAO_CALL,
                       CLAUSULA_LET, MENSAGEM_OBJETO
            parameters : covering, disjunctive
```

Página 88

1. A definição da metaclasse EVENTO não se aplica.

2. No lugar de

```
Metaclass EVENT
   aggregation of "on", NOME_EVENTO, "when", PREDICADO, "do",
                                                 BODY
```

leia-se

```
Metaclass EVENT
   aggregation of ("on"|"instead"), MENSAGEM_OBJETO, (("when"
                              PREDICADO) | {}), "do", BODY
```

3. Acrescente-se a definição

```
Metaclass MENSAGEM_OBJETO
   aggregation of "[", TERMO, METHOD_NAME, (PARAMETERS|{}), "]"
```