UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Evolução de Esquemas em Bancos de Dados Orientados a Objetos Utilizando Versões

por

Miguel Rodrigues Fornari

Dissertação submetida como requisito parcial
para a obtenção do grau de
Mestre em Ciência da Computação

Prof. Lia Goldstein Golendziner
Orientador

Porto Alegre, Setembro de 1993.

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Fornari, Miguel Rodrigues

Evolução de Esquemas em Bancos de Dados Orientados a Objetos Utilizando Versões / Miguel Rodrigues Fornari.— Porto Alegre: CPGCC da UFRGS, 1993.
131 p.: il.

Dissertação (mestrado)—Universidade Federal do Rio Grande do Sul, Curso de Pós-Graduação em Ciência da Computação, Porto Alegre, 1993. Orientador: Golendziner, Lia Goldstein

Dissertação: Banco de Dados, Engenharia de Software
Orientação a objetos, Banco de Dados orientados a objetos, Evolução de esquemas, Versões

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Sistema de Biblioteca da UFRGS

6180

FORNARI, MIGUEL RODRIGUES

EVOLUCAO DE ESQUEMAS EM BANCOS
DE DADOS ORIENTADOS A OBJETOS
UTILIZANDO VERSOES
681.32.072(043)
F727E

INF
1994/250033-9
1994/01/06

MOD. 2.3.2

Jubilate Deo

(Offertorium in Hebdomada quinta Paschae)

Jubilate Deo universa terra;
jubilate Deo universa terra;
psalmum dicite nomini ejus;
venite et audite, et narrabo vobis,
omnes qui temetis a Deum,
quanta fecit Dominus naimae meae,
alleluia.

Aclamai a Deus com alegria

(Ofertório da quinta semana da Páscoa)

Aclamai a Deus com alegria,
aclamai a Deus com alegria, toda a Terra cantai
um salmo ao Seu Nome, vinde e ouvi, e eu
narrarei a vós todos os que temei a Deus,
quantas coisas grandes o Senhor fez em minha alma,
aleluia.

AGRADECIMENTOS

"Agradeço a minha mãe e ao meu pai, e a mãe e ao pai de todos vocês."

Jorge Benjor

Ao povo brasileiro que financiou minha educação em todos os níveis. Espero, sinceramente, ter condições de retribuir as oportunidades oferecidas de modo adequado.

A Profa. Lia Goldstein Golendziner, pela amizade e orientação constante, decisivas para o desenvolvimento e conclusão deste trabalho.

Ao Prof. Flávio Wagner e demais pessoas envolvidas no projeto, pelas diversas discussões que contribuíram para o entendimento e definições relativas ao STAR.

A coordenação, professores e funcionários do CPGCC, que apesar das dificuldades, conseguem manter um ambiente adequado para o desenvolvimento de trabalhos de qualidade.

A CAPES E CNPq, que garantiram minha sustentação econômica.

A Márcia, minha esposa, por tudo.

Aos meus pais, Enio e Selma, que sempre me incentivaram a estudar e perseverar.

Aos meus parentes e amigos, pela convivência agradável e suporte nos pequenos problemas diários.

A Comunidade Neo-Catecumenal, padres e catequistas, por me aceitarem com erros e acertos.

A Deus, que permitiu tudo isso, desde a Criação.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

Este trabalho apresenta um mecanismo para evolução de esquemas para bancos de dados orientados a objetos. A necessidade de alteração do esquema conceitual de dados pode surgir em qualquer momento da vida de um sistema, por motivos como incorporar novas especificações e necessidades do usuário, reaproveitamento de classes em outros sistemas e correção de falhas de modelagem. Uma ferramenta deste tipo deve permitir ao usuário a maior variedade possível de alterações e, ao mesmo tempo, possibilitar um alto grau de independência lógica de dados, para reduzir ao máximo possível a necessidade de alteração dos programas de aplicação que utilizam o esquema.

O modelo de dados utilizado está baseado nos modelos de outros sistemas orientados a objetos, como Orion e O_2. Ele permite a definição de classes, atributos simples e construídos pelo usuário, métodos, como forma de encapsular os objetos e herança múltipla de atributos e métodos para subclasses. Além disso, para manter o histórico de modificações realizadas, versões de instâncias, classes e métodos são utilizadas. Versões de um objeto formam um grafo acíclico, sendo a versão mais recente a "default". Como forma de manter a coerência no uso de versões de diferentes objetos, o conceito de contextos de esquemas é definido.

A proposta baseia-se no conceito de invariantes, condições básicas para a base de dados ser considerada válida e consistente pelo sistema. Invariantes estruturais e comportamentais são definidos e mantidos. Diversas operações que podem ser realizadas sobre um esquema são descritas, detalhando para cada uma as suas opções e efeitos. Alguns mecanismos auxiliares para aumentar a transparência de alterações de esquemas são esboçados.

Como uma aplicação específica do mecanismo genérico apresentado, outro é desenvolvido para o ambiente STAR. Seu modelo de dados e os gerentes de versões e metodologia são explicados, tendo suas características mais relevantes para este trabalho detalhadas. Tomando o esquema de objeto como um esquema de dados

e as tarefas do gerente de metodologias como métodos, o mecanismo também se baseia em invariantes que são utilizados para validar a correção das modificações realizadas, cuja semântica está descrita detalhadamente.

O mecanismo definido revelou-se extremamente flexível e capaz de manter não só o histórico do desenvolvimento de determinada aplicação, como também alternativas de um mesmo sistema que esteja sendo construído utilizando um banco de dados orientado a objetos, tendo atendido satisfatoriamente aos requisitos básicos definidos inicialmente.

PALAVRAS-CHAVE: Orientação a Objetos, Bancos de Dados Orientados a Objetos, Evolução de Esquemas, Versões.

TITLE: "SCHEMA EVOLUTION IN OBJECT ORIENTED DATA BASES USING VERSIONS"

ABSTRACT

This work presents a schema evolution mechanism, based on an object oriented data model. Conceptual schema modifications are needed at any moment in the life cycle of a system, for example, to incorporate new specifications and users' solicitations, to reuse classes developed for other system and to correct modeling errors. This mechanism has to allow a great number of different operations and, at the same time, a high data logic independence to reduce the number of changes in applications programs.

For this proposal we are considering an object oriented data model, similar to those existing in Orion and O_2. Class definitions, simple attributes and attributes constructed by the user, methods to encapsulate objects and multiple inheritance of attributes and methods to subclasses are allowed. Instances, classes and methods are versionable. Connected directed acyclic graphs organize the versions of an object. There is one current version, which either is the most recent (the default) or one defined by the user. Schema contexts are introduced to keep track of the correspondence that exists among all the versions created, assuring the selection of a method version adequate for a version instance.

The mechanism is based on schema invariants, that are basic conditions that always must be satisfied in order to insure that the schema is in a correct state. Structural and behavioral invariants are defined and checked by the system. The designer can use a complete set of operations to change the schema. The semantic of all operations is described, with its options and effects. Some auxiliary mechanisms are incorporated to facilitate schema change transparency.

As an application, a generic mechanism for schema evolution is developed to the STAR framework. The data model, version and methodology managers of STAR are explained. The mechanism is based on invariants to validated changes considering an object schema as conceptual schema and the methodology manager tasks like methods .

The mechanism is extremely flexible and capable of maintaining the history of schema development and alternatives of classes, methods and instances' descriptions. The previously defined characteristics are allowed in a satisfactorily way, resulting in a very useful tool for software design.

KEYWORDS: Object Orientation, Object Oriented Data Bases, Schema Evolution, Versions.

1 INTRODUÇÃO

O paradigma de orientação a objetos está sendo cada vez mais utilizado e pesquisado na área de bancos de dados. Bancos de dados orientados a objetos apresentam um modelo de dados complexo, baseado em conceitos de abstração e em métodos para a manipulação dos objetos, o que torna possível reconhecer parte da semântica dos dados armazenados. Como este paradigma tem por característica permitir a construção dos objetos em paralelo com o desenvolvimento das operações que podem ser realizadas sobre as classes, através da especificação de métodos, verificou-se ser necessária a introdução de mecanismos que permitam a alteração do esquema conceitual das classes/tipos presentes.

Nestes sistemas, a alteração de esquemas torna-se normal, contrariamente às aplicações tradicionais, pois o projetista certamente realiza várias modificações na modelagem dos dados que implicam operações de correção sobre a definição do esquema [BER 91]. Nos sistemas relacionais, que suportam atualmente a maioria das aplicações convencionais, foram encontradas poucas propostas expandindo as definições iniciais [AST 76, AST 79, DAT 86] dos comandos em SQL `CREATE TABLE`, `EXPAND TABLE` e `DROP TABLE` [LAI 79, MCK 90, ROD 92], embora as aplicações tenham uma certa necessidade de alteração das tabelas armazenadas na base de dados. Mesmo na mais recente norma de definição do comitê da ISO para SQL [MEL 92], que já inclui a possibilidade de criação de esquemas compostos por um conjunto de tabelas, visões ("views"), domínios, valores "defaults" para campos e restrições de integridade para tuplas e campos, as possibilidades de mudança são pequenas. É permitido apenas acrescentar e remover novas tabelas, campos a tabelas já existentes, valor "default" para um campo, restrição de integridade, domínios e visões.

Necessidades semelhantes também são encontradas em sistemas para modificação dinâmica de programas em execução ("Updating Systems"). Segal & Frieder [SEG 93] analisam vários destes sistemas e observam que a maioria permite apenas a troca da implementação de um procedimento ou de um tipo abstrato de

dados. Para suprir esta deficiência, propõem a definição de interprocedimentos, que possuem uma interface idêntica àa anterior e realizam a conversão dos parâmetros para a nova chamada. Porém, este método pode resultar em um acúmulo de interprocedimentos, sobrecarregando o sistema.

A necessidade de alteração do esquema de uma base de dados pode resultar de diversos fatos e surgir em qualquer momento da vida do sistema, desde a fase de projeto até a de produção:

- Alterações na especificação do projeto do sistema, conseqüência de uma alteração externa ao ambiente computacional. Por exemplo, uma modificação no sistema de cálculo do imposto de renda, decretada pelo governo.

- Surgimento de novas necessidades do usuário, que obriguem a remodelação do esquema para atender a consultas anteriormente não previstas. Por exemplo, incluir na lista de funcionários da empresa uma relação de empresas concorrentes nas quais tenha trabalhado.

- Reaproveitamento de esquemas ou classes já existentes e testadas em algumas aplicações como base para o desenvolvimento de novas aplicações, onde não atendem completamente a especificação, necessitando pequenas alterações para isto. Este reaproveitamento é extremamente desejável como forma de redução de custos no desenvolvimento de sistemas computacionais.

- Quando houver a utilização de uma base de dados em um sistema federado, onde incompatibilidades de nomes ou domínios podem ocorrer, obrigando a realização de modificações para resolver os problemas identificados.

- Alterações no sistema para corrigir falhas ocorridas em qualquer das suas etapas de desenvolvimento. Considerando que no paradigma de orientação a objetos, o mundo real é definido tanto estrutural como com-

portamentalmente, com a etapa de especificação confundindo-se com a implementação, o surgimento de falhas sempre ocorre.

Estas situações também ocorrem nas aplicações comerciais já existentes, que estão baseadas em modelos relacionais, porém as pesquisas têm se concentrado em propostas de evolução de esquemas para bancos de dados orientados a objetos, onde os problemas envolvidos são, naturalmente, mais complexos devido a maior quantidade de conceitos semânticos mantidos pelo banco de dados.

1.1 Conceitos Básicos do Modelo Orientado a Objetos

Antes de abordarmos especificamente a evolução de esquemas em bancos de dados orientados a objetos, é necessária uma rápida explicação dos principais conceitos deste paradigma. Embora não haja uma definição formal para o paradigma, alguns conceitos básicos estão incorporados a quase todos os sistemas que o adotam, sejam linguagens de programação ou bancos de dados. Uma análise mais abrangente das variações existentes, do ponto de vista da área de Banco de Dados, pode ser encontrada em [MAT 93]. Os princípios fundamentais são [BER 91, GRO 92]:

- Cada entidade existente no mundo real é modelada em um **objeto**, que possui existência e identidade própria, manifestada através de um identificador de objeto, definido pelo sistema, e único na base de dados.
- Cada objeto está encapsulado em sua estrutura e comportamento. O primeiro é descrito através de **atributos**, com o conjunto de valores dos atributos representando o estado do objeto. Estes valores podem ser identificadores de outros objetos, permitindo a definição de objetos complexos através de agregação. O segundo consiste de um conjunto de **métodos**, isto é, procedimentos e funções que podem ser executadas sobre o objeto.

- O estado de um objeto pode ser consultado e alterado exclusivamente através do envio de **mensagens** ao objeto, que acionam o método correspondente.

- Objetos compartilhando a mesma estrutura e comportamento são agrupados em **classes**, que representam um modelo para um conjunto de objetos similares. Cada objeto é instância de uma classe.

- Uma classe pode ser definida como uma especialização de uma ou mais classes. Neste caso, ela será uma **subclasse** de suas **superclasses**, de quem herdará estrutura e comportamento, de acordo com o conceito de generalização.

Através de uma classe, é possível gerar objetos que respondam a todas as mensagens definidas para a classe, pois todas as suas instâncias possuem a mesma estrutura e comportamento.

O conceito de generalização provê um meio para tornar concisa a descrição de propriedades do mundo real, pois permite que uma classe, chamada subclasse, seja construída a partir de outra já existente, a superclasse, cujas definições herda [SMI 77]. A subclasse pode possuir propriedades específicas suas e redefinir características herdadas das superclasses. Organiza-se, através deste conceito, uma hieraquia multinível de classes. Alguns sistemas restringem a hierarquia à forma de árvore, cada subclasse possuindo apenas uma superclasse. Associado a este conceito, está o mecanismo de herança de atributos e métodos. No caso de haver apenas uma superclasse, chama-se **herança simples**. Quando se permite a formação de um grafo acíclico de classes, onde cada classe pode possuir uma ou mais superclasses, chama-se **herança múltipla**.

Na herança múltipla, problemas de conflitos de nomes podem ocorrer, pois propriedades diferentes podem estar definidas com nomes iguais. Por exemplo, a classe *Barcos* pode ser definida como subclasse das classes *Veículos_Motorizados* e *Veículos_Aquáticos*. Na classe *Veículos_Motorizados* pode haver uma propriedade

chamada *Velocidade_Máxima* expressa em quilômetros por hora, enquanto na classe *Veículos_Aquáticos* pode haver outra, também chamada *Velocidade_Máxima*, mas expressa em nós, ou seja, duas propriedades com nomes iguais e domínios diferentes serão herdadas pela classe *Barcos*. Há duas abordagens utilizadas para resolver conflitos de nomes: uma obriga o usuário a sempre especificar a qual dos possíveis candidatos está se referindo [LEC 88] e outra estabelece algumas regras que o sistema segue para selecionar um dos possíveis nomes [BAN 87].

Toda visão que o usuário possui da estrutura dos objetos é a fornecida pelos métodos definidos para a classe do objeto. Em geral, métodos são compostos pela sua assinatura, isto é, a relação dos seus parâmetros, e o seu corpo, o código executável da operação. Métodos também são herdados para as subclasses, onde podem ser redefinidos. Estes métodos são escritos em uma linguagem específica de cada sistema, que pode ser compilada e ligada a programas em linguagens de programação como C, Pascal e Lisp. Como os métodos são utilizados para organizar o comportamento de uma classe, que está descrito no código fonte de cada método, uma parte importante da semântica de cada objeto/atributo é retirada do programa de aplicação e passa a ser armazenada também no sistema. Este fato torna o paradigma orientado a objetos, como base para um banco de dados, vantajoso em relação ao relacional, pois reduz sensivelmente a possibilidade de mudanças nas aplicações, como destaca [VEN 91].

1.1.1 Evolução de Esquemas

Como a complexidade dos sistemas desenvolvidos utilizando orientação a objetos é maior, pois um grande número de conceitos são utilizados para modelar a aplicação, torna-se importante a existência de uma ferramenta automática que permita a realização de operações de evolução de esquemas. Esta ferramenta apresenta alguns requisitos básicos:

- Manter a integridade global do sistema, ou seja, a integridade dos aspectos estruturais e comportamentais. Considerando o número de relacionamentos de abstração existentes em um esquema de objetos, o projetista certamente necessita de um auxílio automático para manter a integridade da base de dados.

- Preservar a correção de programas que já estão em funcionamento durante e após a alteração do esquema. Programas testados não podem se tornar incorretos, sob pena de todo sistema se tornar desacretidado por ser inseguro.

- Realizar a modificação enquanto o sistema está em operação. Algumas aplicações devem se manter em funcionamento permanente, não sendo viável técnica e economicamente, uma interrupção do sistema. Assim, os efeitos da alteração sobre a base de dados e as aplicações que a utilizam deve ser, se possível, imperceptível para o usuário e, principalmente, para o desempenho global do sistema.

- Minimizar a necessidade de intervenção humana, pois até mesmo o mais meticuloso projetista de software pode realizar alterações incorretas.

- A ferramenta deve possuir flexibilidade, permitindo ao usuário a maior variedade possível de alterações.

- Facilidade de uso. Neste aspecto, a existência de uma interface gráfica seria extremamente útil, pois linguagens textuais, em geral, são mais complexas para o aprendizado e não fornecem uma visão geral do esquema.

Uma vez atendidos estes requisitos, o mecanismo de evolução de esquemas resultante será uma ferramenta extremamente poderosa e útil no auxílio ao projetista da aplicação, que tornará o desenvolvimento de aplicações sobre o banco de dados uma tarefa mais rápida e confortável. Porém, o atendimento de todos os requisitos torna a implementação do sistema uma tarefa extremamente complexa.

Para cada uma das diversas operações de evolução de esquema possíveis, uma semântica deve ser estabelecida para especificar como e se as instâncias já armazenadas na base de dados serão afetadas e garantir a correção do esquema resultante. Dois tipos de integridade podem ser definidos para um esquema: **estrutural** e **comportamental** [LEC 88].

Por **integridade estrutural** entende-se que há correção nos grafos de generalização e agregação. Ou seja, não há ciclos no grafo de generalização, todos os conflitos de nomes estão resolvidos e propriedades redefinidas nas subclasses estão coerentes com a definição herdada.

Por **integridade comportamental** entende-se que os métodos estão corretos. Suas assinaturas utilizam parâmetros com tipos corretos e existentes, mensagens são enviadas apenas para métodos públicos, o valor de retorno tem o tipo certo, apenas objetos da classe para a qual está definido são alterados e os atributos e outros métodos referidos existem.

Outro tipo de integridade pode ser acrescentado, como uma especialização do conceito de integridade estrutural, a **integridade de instanciação** [FOR 92]. Por integridade de instanciação entende-se que todas as instâncias, quando consultadas ou alteradas, estarão de acordo com a definição atualmente válida. De um modo geral, esta integridade está englobada na estrutural, pois instâncias são as folhas do grafo de generalização, mas sua definição será útil para explicar os mecanismos de alteração das instâncias. É importante notar que por esta definição não é necessário manter as instâncias constantemente atualizadas.

Após cada alteração realizada sobre o esquema, o sistema deve verificar se os três tipos de integridade foram atendidos. Em geral, a integridade estrutural é mantida através de invariantes para o esquema. Estes invariantes asseguram, de acordo com o modelo de dados particular, que a parte estática do esquema está correta. Os principais sistemas, como o ORION [BAN 87], GemStone [BUT 91] e O_2 [LEC 88], utilizam este mecanismo.

Quanto à integridade comportamental, é muito difícil ter um controle total das operações, pois a semântica de cada operação está inserida na sucessão de instruções que compõem o código do método. É possível manter uma lista de atributos e outros métodos utilizados, mas pouca informação sobre a atuação do método é obtida deste modo. Porém, isto já é suficiente para indicar alguns problemas que podem ocorrer. Por exemplo, caso o tipo de um atributo seja redefinido na subclasse, os métodos que utilizavam-no precisam ser revistos para confirmar se as operações realizadas sobre este atributo continuam corretas. Os métodos atingidos podem ser identificados pelas listas de atributos e indicados para o projetista. Este pode corrigi-los e recompilá-los. É necessário notificar o projetista, pois não é possível realizar uma correção automática, já que apenas ele possui completo domínio da semântica do código, e, principalmente, domínio de como alterá-la corretamente.

A integridade de instanciação é mantida através de três mecanismos:

- **Propagação Diferida**: A alteração realizada a nível de esquema é postergada até que o objeto seja consultado pelo usuário. Esta proposta exige que o sistema seja capaz de armazenar diferentes representações de objetos de uma mesma classe e obriga, a cada acesso, que seja realizada uma verificação da correção de sua representação [LEC 88, RUG 91].

- **Propagação Imediata**: A alteração é imediatamente propagada para todas as instâncias da classe. Exige acesso a todos os objetos da classe para atualização. Seu desempenho é proporcional ao número de objetos armazenados, o que pode tornar inviável sua utilização para um número muito grande de objetos. A consulta é simples e imediata [MAT 91, PEN 87].

- **Visões**: O sistema jamais retira uma informação da base de dados, apenas altera a visão atual que o usuário possui sobre a classe. Acréscimos são realizados imediatamente sobre os objetos. Torna as consultas mais lentas, mas facilita operações de recuperação de dados removidos [BAN 87a, SKA 86]. Visões também são chamadas de **filtros**.

1.2 Versões

A natureza do processo de desenvolvimento de produtos, sejam sistemas computacionais, digitais, prédios ou automóveis, requer a possibilidade de armazenar e gerenciar diversos estados de um mesmo objeto, chamados versões [DIT 87, FAU 91, KAT 90]. A tarefa do projetista baseia-se no desenvolvimento de algumas idéias que poderão ser demonstradas como ruins, bem como a realização de várias alternativas de projeto para verificar qual obtém melhores resultados.

Vários SGBDOO [BAN 87, BER 88, BJÖ 89, BUT 91, CEL 90, DAV 86, FAU 91, LAM 91, SKA 86, WIL 90] suportam a idéia de versionamento de objetos. O objeto não possui uma representação única na base de dados. Os diversos estados, chamados **versões**, estão armazenados em uma estrutura, chamada **grafo de derivação**, que permite o ordenamento de acordo com o instante de tempo em que a versão foi inserida na base de dados e, segundo o critério do projetista, de qual versão anterior está é derivada. A estrutura pode ser linear, em árvore ou um grafo acíclico.

Versões podem ser utilizadas em um banco de dados com três objetivos específicos [BJÖ 89]:

1. Capturar a história de um objeto. Além do aspecto temporal, que pode ser expresso através de uma sucessão linear de versões, pode-se manter derpendências entre versões diferentes. Neste caso, é necessário um grafo acíclico, pois de uma versão pode-se derivar duas alternativas diferentes ou combinar duas ou mais versões em uma única versão.

2. Lidar com os problemas de evolução de esquemas. Quando alterações são realizadas em uma classe, todas as instâncias e aplicações devem ser adaptadas para a nova representação. Esta reorganização pode requerer uma grande quantidade de modificações nos dados armazenados e em problemas de desempenho e restrição de acesso.

3. Aumentar o grau de concorrência do sistema, através de bloqueios a uma versão do objeto somente, permitindo acesso concorrente a outras versões do mesmo objeto, e no mecanismo de recuperação de falhas, retornando a um estado anterior da base de dados.

O armazenamento da história do objeto é especialmente útil para o projetista que está trabalhando no desenvolvimento de um projeto. A história deste objeto é formada não só pelas alterações provenientes de modificações em suas características, mas também em sua propriedades descritas no esquema. Este mecanismo evita a remoção de um estado anterior do projeto através de alterações, permitindo retornar a ele se algumas alterações realizadas revelarem-se, posteriormente, inadequadas.

O retorno a um estado anterior da modelagem dos dados pode ser realizado, de forma consistente, através da seleção de uma versão mais antiga. Para isto, basta garantir que no momento em que foi armazenada, a versão estava correta, com todos os seus relacionamentos atendendo às especificações contidas no esquema. Assim, a correção de erros cometidos durante o projeto fica facilitada, pois é possível retornar a um estado correto e retomar o desenvolvimento a partir daquela etapa.

A reusabilidade de objetos também é incrementada, pois estágios iniciais de classes e objetos podem se adequar perfeitamente a uma especificação de um novo sistema, pois, eventualmente, o estado atual já conta com detalhes supérfluos para ela. Assim, o estado anterior, menos detalhado, pode ser reaproveitado e trabalhado na modelagem do esquema da nova aplicação.

Em aplicações de projeto, tipicamente, vários projetistas compartilham um mesmo objeto, podendo manter diferentes bloqueios sobre ele. Uma alteração de esquema, com conseqüências sobre todos objetos da classe e das subclasses, pode tornar-se impossível de ser realizada por um dos projetistas, se ela necessitar que não haja nenhum bloqueio de outros usuários sobre todos os objetos afetados.

Outro fato que pode ocorrer é dois projetistas realizarem operações conflitantes de evolução de esquema sobre uma classe em um determinado instante. Por exemplo, alterar o domínio D de um atributo A para dois domínios diferentes D' e D''. A priori, não há como estabelecer que uma das alterações apenas é a correta, ou que possui prioridade sobre a outra. Neste caso, a utilização de versões é extremamente útil, por que possibilita a manutenção de diferentes alternativas de um mesmo objeto. Assim, ambas as operações seriam permitidas e duas novas versões, com as respectivas descrições corretas do ponto de vista individual dos projetistas, embora conflitantes, podem ser mantidas na base de dados, aumentando o nível de concorrência. Paralelamente, um sistema de direitos de acesso pode ser acoplado, isolando, numa etapa inicial, as visões individuais de cada projetista.

A recuperação de falhas em um ambiente com versões também se torna uma tarefa menos crítica, pois versões anteriores consistentes do objeto podem ser encontradas na base de dados já consolidada. Alguns trabalhos com propostas semelhantes podem ser encontradas em [BER 83, BJÖ 89].

Comparando estas facilidades oferecidas pelo uso de versões com os requisitos definidos anteriormente para uma ferramenta de evolução de esquemas, percebe-se a utilidade do uso de versões neste mecanismo.

1.3 Objetivos do Trabalho

O objetivo deste trabalho é definir uma proposta que permita a evolução de esquemas em SGBDOO, sem perda de integridade do esquema e procurando manter o conceito de independência de dados, isto é, não afetando as aplicações já existentes.

Visando a alcançar este objetivo, um mecanismo de versionamento será utilizado, para manter diferentes estados corretos do esquema. Relacionamentos entre estas versões deverão ser mantidos como forma de armazenar configurações corretas do esquema.

As conseqüências da alteração do esquema sobre os objetos instanciados na base de dados e os métodos definidos deverão ser estudadas e os problemas identificados resolvidos adequadamente.

As operações necessárias para a realização do mecanismo proposto deverão ser especificadas.

Como forma de validar o mecanismo sugerido, as necessidades específicas do ambiente STAR serão modeladas com a proposta realizada.

1.4 Organização do Trabalho

Este trabalho está dividido em sete capítulos. Neste capítulo inicial, alguns conceitos básicos de orientação a objetos, evolução de esquemas e versões foram definidos, de acordo com a sua utilização neste trabalho, pois sobre eles ainda não há um consenso formado. No capítulo seguinte, uma análise dos principais sistemas que também suportam evolução de esquemas é realizada. No capítulo 3, o modelo de dados genérico que será utilizado é apresentado. No quarto capítulo, os mecanismos para a realização de evolução de esquemas sobre um SGBDOO é definido. Na capítulo 5, o modelo de dados do ambiente STAR é explicado e no sexto capítulo, a aplicação do modelo genérico para o específico existente no STAR é descrita. Esta aplicação define um esquema para evolução de esquemas no ambiente STAR que está sendo implementado. No sétimo capítulo, os resultados obtidos são explicados e comentados e algumas indicações para o seguimento deste trabalho são traçadas.

2 REVISÃO DA LITERATURA

Este capítulo descreve algumas propostas de evolução de esquemas em bancos de dados que representam as mais importantes contribuições na área com bibliografia disponível. Para cada trabalho apresentado, o modelo de dados e as características relevantes do sistema são citadas. Posteriormente, descreve-se as possibilidades de evolução permitidas para o esquema e realiza-se uma análise crítica das mesmas. Uma comparação mais extensiva pode ser encontrada em [FOR 92].

2.1 ENCORE

ENCORE é um SGBD desenvolvido na Brown University [SKA 86, ZDO 86] que suporta os conceitos do paradigma de orientação a objetos e versionamento.

Um objeto é uma abstração de dados com uma representação interna conhecida apenas pelo seu tipo e uma interface composta por propriedades, operações e restrições. O conjunto de instâncias de um tipo forma uma classe. A descrição dos tipos também está armazenada em objetos do Banco de Dados.

Hierarquias de tipos podem ser criadas, definindo supertipos e subtipo. As propriedades, operações e restrições de um supertipo são herdadas pelos seus subtipos. Herança múltipla é suportada, ficando definido um grafo acíclico. A raiz deste grafo é o tipo *Entity*, pré-definido. Propriedades podem ser redefinidas a nível de subtipo, mas não podem ser removidas, expandidas ou contrariadas.

Uma função de pertinência de objeto a classe pode ser especificada. Genericamente, esta função está definida da seguinte forma:

$$f_{Classname}(o) = \begin{cases} verdadeiro & \text{se } o \in Classname \\ falso & \text{caso contrário} \end{cases}$$

As condições para esta função podem ser arbitrariamente definidas pelo usuário e envolver valores de propriedades do tipo. Por exemplo, a função de pertinência do tipo Cores_de_Fiat pode ser especificada como sendo composta por objetos da classe Cor e com um nome igual a preto ou vermelho, como segue:

$f_{Cores\_de\_Fiat}(o) = \{\ type(o) = Cor \wedge (value(nome\_da\_cor(o))) = preto \vee (value(nome\_da\_cor(o))) = vermelho\ \}$

2.1.1 Evolução de Esquemas

A alteração na estrutura de um tipo traz conseqüências também para os objetos armazenados no banco de dados. Algumas operações que podem ocorrer estão listadas abaixo, o modo como afetam os objetos está descrito a seguir.

- Adicionar ou remover um tipo.
- Mover um tipo para uma nova posição no grafo de generalização.
- Modificar um tipo.
  - Adicionar ou remover propriedades, operações ou restrições definidas para o tipo.
  - Modificar propriedades, operações ou restrições.

Um mecanismo combinando versões de objetos e tratadores de erros foi proposto para solucionar os problemas que uma alteração do tipo resulta. Um conjunto de versões é uma coleção ordenada cronologicamente de todos os estágios de um objeto. Este conjunto é inicializado com a primeira definição do objeto inserida na base de dados, após cada alteração do tipo é aumentado e retirado da base de dados na remoção deste. A modificação de um tipo requer a escolha de uma versão existente na base de dados e criação de uma nova versão baseada na anterior.

Cada instância é ligada a uma versão de tipo, contendo as propriedades definidas nesta versão. Eventualmente, uma função pode tentar utilizar uma pro-

priedade que não está definida no objeto ou que contém um valor inválido para a versão atual de tipo. Para estes casos, tratadores de erros são utilizados para manipular todas as versões de um determinado tipo que tenha sido alterado. Estes tratadores de erros filtram estas ocorrências, retornando um valor coerente, definido pelo projetista. Tratadores de erros são herdados para as subclasses, onde podem ser redefinidos. A cada nova versão de tipo gerada, o projetista deve incluir novos tratadores de erro específicos para a alteração realizada. Estes tratadores de erros visam permitir que uma função definida para certa versão de tipo possa ser aplicada, de forma transparente, em instâncias pertencentes a outras versões do tipo.

Por exemplo, na versão 2 de *Carro*, como definido abaixo, o conjunto de valores aceitáveis para a propriedade *Cor* foi restrito somente a *Preto*. Como na versão 1, outros valores eram aceitos, o tratador altera estes valores para *Preto* no momento em que ele é consultado por alguma função, sem ser necessário a alteração da instância armazenada na base de dados.

```
DEFINE TYPE Carro.V1
  SUPERTYPES: Veiculo.V1
  PROPERTIES:
    Cor: {Verde, Azul, Preto}
```

```
DEFINE TYPE Carro.V2
  SUPERTYPES: Veiculo.V1
  PROPERTIES:
    Cor: {Preto}
  HANDLERS:
    if value(Cor) <> Preto
       return(Preto)
```

A proposta apresenta algumas qualidades, como a utilização de um mecanismo de versões para armazenar as alterações ocorridas em objetos. Os tratadores de erros não resolvem a questão de acessos a objetos de acordo com um esquema já alterado e transferem para o projetista o ônus de adequar o sistema para permitir a sua utilização. O mecanismo apenas fornece ao projetista um modo para evitar uma interrupção do programa.

2.2 Proposta de Zdonik

Zdonik complementa as possibilidades de evolução de esquemas em um modelo de dados que suporta polimorfismo e herança múltipla [ZDO 90]. Não há um SGBDOO para o qual a proposta tenha sido definida, embora possa se supor que o trabalho anterior no ENCORE tenha forte influência. **Polimorfismo** permite que um objeto x seja criado como instância de um conjunto S de tipos, por exemplo, uma determinada instância chamada *Gurgel* pode pertencer a *Carros*, *Veículos* e *Objetos_Móveis*. Dois conjuntos de tipos podem ser definidos pelo usuário:

- **Tipos Essenciais**: Se pensarmos como um objeto evolui no mundo real, veremos que algumas transações jamais ocorrem. Por exemplo, uma pessoa não se transforma em carro. Isto ocorre porque objetos, por mais que sejam alterados, não deixam de pertencer a determinada classe genérica, isto é, algumas características não são perdidas nunca. Uma *Pessoa* pode passar de *Estudante* para *Professor* e, depois, para *Aposentado*, mas sempre será uma *Pessoa*. Definindo tipos essenciais de uma classe, estabelece-se uma restrição às operações que podem ser realizadas sobre objetos a ela pertencentes.

- **Tipos Excludentes**: Do mesmo modo, pode-se definir um tipo T como sendo excludente se a associação do objeto ao tipo só for permitida no momento de sua criação. T é chamado excludente porque ao mover um objeto x de um tipo R para outro S, é ilegal mover para um que possua T como supertipo, ou seja, T e seus subtipos estão excluídos do conjunto de possíveis novos tipos.

Com estes mecanismos pode-se estabelecer uma seqüencia pré-definida de alterações possíveis para o objeto. Por exemplo, forçar um objeto a ser inserido como *Criança*, passar a *Estudante*, *Professor* e *Aposentado*, sem jamais deixar de ser *Pessoa*.

2.2.1 Problemas na Alteração de Tipos

Com a alteração da definição de tipos dois problemas surgem quanto ao funcionamento do sistema:

1 - Manter os velhos programas funcionando com novas instâncias do tipo.

2 - Permitir que novos programas reconheçam velhas instâncias.

Uma solução que resolva estes problemas, chamada *transparência de alteração de tipos* não foi definida pelo autor. Assegurar que o esquema está consistente não garante, de nenhuma forma, o funcionamento correto dos programas de aplicação. Diversas alternativas têm sido estudadas, mas não resolvem o problema completamente.

- **Conversão dos Dados**: Converter todas as instâncias do tipo para a nova definição pode comprometer o desempenho do sistema, ou não ser possível, devido ao grande número de instâncias, e exige a conversão de todos os programas antigos.

- **Tratamento de Exceções**: As versões antigas não são removidas. O projetista pode acrescentar tratadores de exceção sempre que um programa realizar uma operação inválida devida à alteração de algum tipo. Obriga o projetista a definir tratadores de exceção a cada alteração e pode resultar em uma grande quantidade de tratadores para uma mesma classe, sobrecarregando o sistema.

- **Múltiplas Visões**: Nesta abordagem, um objeto é visto como sendo um conjunto de visões, cada uma representando um estado específico no histórico de evolução do objeto. Este proposta é cumulativa, nenhum objeto é alterado, apenas uma nova visão é sobreposta à antiga quando da execução da operação.

Este trabalho é complementar a outros já realizados, inclusive pelo autor. Estende os mecanismos de evolução de esquema já existentes definindo tipos essenciais e excludentes, que representam restrições na evolução do esquema, acrescentando mais semântica ao SGBDOO. A idéia de manter a transparência de alteração de tipos, citada, é importante pois visa evitar a modificação de programas já em funcionamento.

2.3 GemStone

GemStone [MAI 86, PEN 87, BUT 91] é um SGBDOO desenvolvido com o objetivo de combinar as facilidades das linguagens orientadas a objetos com as capacidades de SGBDs já existentes, atualmente comercializado pela Servio Corporation. Pretende ser uma ferramenta que permita o desenvolvimento de protótipos rapidamente implementáveis e de fácil manutenção. Para facilitar o processo de prototipação, um esquema de modificação de classes foi definido para o sistema.

GemStone suporta quatro tipos básicos de objetos: **auto-identificáveis**, para objetos atômicos como `SmallInteger`, `Character`, `Boolean`; **byte**, para seqüências de bytes como `String`, `DateTime`, `Float`; **ponteiros**, para referenciar outros objetos, como `Ponto`, `Empregado`, `Carro` e **coleções** para armazenar objetos de várias classes. A herança de definição de atributos existe da superclasse para as subclasses. Métodos devem ser escritos para possibilitar a realização de consultas e atualizações sobre a base de dados.

Objetos podem ser versionados visando permitir o seu desenvolvimento por dois ou mais projetistas em paralelo, que consolidariam suas modificações em diferentes versões derivadas de uma mesma versão anterior. Em um segundo momento, um projetista pode reunir duas ou mais versões em uma nova versão. Deste modo, um grafo acíclico de versões pode ser obtido para determinado objeto.

2.3.1 Evolução de Esquemas

Seis invariantes foram definidos para o GemStone. Todas as operações de alteração do esquema devem manter inviolados estes invariantes ao final da operação para manter a correção do esquema:

- **Representação**: Todas as instâncias de uma determinada classe possuem a representação definida para a classe.
- **Hierarquia de Classes**: Todas as classes definidas pertencem a uma hierarquia em árvore, isto é, possuem uma e somente uma superclasse, com exceção da classe `object`, pré-definida, que é a raiz da árvore.
- **Herança Total**: A representação de instâncias determinada para uma classe é herdada completamente por todas as suas subclasses. Não pode haver atributos definidos com o mesmo nome em uma classe e em suas subclasses.
- **Herança de Restrições**: A restrição de uma variável herdada deve ser consistente com a restrição definida na superclasse, isto é, deve ser uma restrição igual ou mais forte que a existente no nível superior.
- **Validade Referencial**: Todas as referências a objetos devem estar corretas, isto é, apontar para um objeto armazenado no sistema que atenda à restrição imposta para o atributo ou serem nulas.
- **Perda de Informação**: A informação sempre é preservada. Todas referências a um objeto podem ser desfeitas, mas o objeto persiste na base de dados. Não há operação para remover um objeto. Alterações em uma classe podem ser realizadas a qualquer momento, mesmo que ocasionem perda de informação. Um relatório é enviado para os proprietários dos objetos pertencentes à classe alterada para estes decidirem se a alteração deve ser desfeita e os dados recuperados.

Diversas operações sobre a definição de classes estão disponíveis para o usuário. Não foram exploradas todas as possibilidades existentes e, mesmo que de modo informal, a semântica das alterações foi definida. Todas as alterações refletem-se imediatamente sobre os objetos pertencentes à classe modificada. As seguintes operações estão disponíveis:

- Alterar o nome de um atributo
- Adicionar um atributo
- Remover um atributo
- Modificar a restrição de um atributo
- Adicionar uma classe
- Remover uma classe
- Indexar uma classe
- Desindexar uma classe

O método da emissão de relatórios para usuários proprietários de objetos atingidos por alterações originárias das operações descritas permite a sua utilização mesmo que um dos objetos atingidos pertença a outro usuário. Porém, isto obriga o proprietário a realizar a operação inversa caso queira reinserir dados removidos. Conflitos surgirão sempre que alguns usuários concordarem em realizar a alteração e outros não.

Para evitar problemas resultantes de alterações concorrentes, um mecanismo de bloqueio para objetos foi implementado. Outro mecanismo para permitir alterações simultâneas em uma classe foi definido. A idéia de existir uma transação de alteração, para encapsular diversas alterações sem necessidade de modificar imediatamente os objetos a cada uma delas, é sugerida. Outro item abordado é quanto à verificação de tipos ser realizada em tempo de execução ou compilação, levando em

conta a possibilidade de alterações ocorrerem sobre classes que invalidem o código executável gerado. A solução encontrada é ter ferramentas para ambos os casos, permitindo ao usuário a definição da alternativa que atende melhor suas necessidades.

Algumas operações realizadas sobre uma classe não são propagadas para as subclasses. Por exemplo, ao se remover um determinado atributo de uma superclasse, onde ele está definido, todas as subclasses que herdavam esta definição, mantêm o atributo, repetido localmente em cada uma delas. Outras operações, como renomear um atributo, são propagadas para as subclasses, tornando o sistema um pouco confuso.

O trabalho é facilitado devido ao modelo de dados não permitir herança múltipla, como em outros SGBDOO, evitando, assim, uma série de considerações sobre conflito de herança.

A utilização de invariantes fornece um modo de verificação da validade das alterações simples e eficiente, eventualmente restritivo. A idéia de uma transação de alteração poderia servir para relaxar estes invariantes, que seriam verificados somente ao final da transação. Além disso, apenas versões de instância são permitidas, ou seja, a modificação em uma classe significa a perda do seu estado anterior.

2.4 ORION

Desenvolvido para servir como apoio para aplicações em CAD/CAM, Inteligência Artificial e Automação de Escritório, o SGBDOO ORION pretende armazenar grandes quantidades de informação rapidamente recuperáveis [BAN 86, WOE 86, BAN 87, KIM 90]. ORION inclui um subsistema de controle e gerenciamento de versões, objetos multimídia e alterações de esquemas conceituais.

ORION suporta o conceito de objetos, que devem pertencer a uma classe. O seu comportamento pode ser descrito em métodos. A descrição da classe também

está armazenada em objetos, chamados meta-objetos. Um método pode ser ativado por uma mensagem. Para os meta-objetos já há métodos definidos.

Um objeto é formado por variáveis de instâncias (atributos) que podem ser definidas como sendo um conjunto de objetos pertencentes a uma classe, um ponteiro para um objeto ou um valor primitivo. Um atributo pode ser **compartilhado** ou **default**. Sendo compartilhado, todas as instâncias da classe assumem um mesmo valor para o atributo; sendo "default", todas as instâncias da classe cujo valor do atributo não for especificado assumem um valor pré-definido. A estrutura de uma classe é herdada para suas subclasses, que podem especificar novas atributos. Uma subclasse pode possuir várias superclasses, definindo um grafo acíclico de objetos.

Objetos de uma classe podem ser declarados como sendo um agregado, composto por objetos pertencentes a outras classes. O objeto componente pode ser dependente do seu agregado, ou seja, sua existência dependerá da existência do agregado, ou independente. Um agregado pode ser composto por objetos dependentes e independentes. Componentes podem ser exclusivos ou compartilhados. A relação de composição é definida por atributos, ditos **de composição** (no original, "composite"), cujo domínio é uma classe.

Como forma de armazenar representações diferentes de um mesmo objeto, este pode ser versionado. Versões podem ser transitórias, alteráveis, ou estáveis. As promoções de transitórias até estável podem ser acionadas pelo usuário ou automaticamente pelo sistema.

2.4.1 Evolução de Esquemas

A proposta para evolução de esquemas no ORION [BAN 87, BAN 87a, KIM 89] baseia-se em invariantes e regras de resolução de ambigüidades. Um editor gráfico foi desenvolvido para auxiliar o usuário nas alterações de esquemas. Os invariantes no ORION são as seguintes:

- **Grafo Acíclico**: O grafo de relacionamentos de generalização deve ser conexo e acíclico. Há apenas uma raiz chamada OBJECT, pré-definida.

- **Nomes Únicos**: Todas as classes possuem nomes únicos. Os atributos e métodos de uma classe, herdados ou não, possuem nomes únicos.

- **Origens Distintas**: Todos atributos e métodos de uma classe, se possuírem o mesmo nome são de origens distintas, logo, conflitos de nome devem estar resolvidos.

- **Herança Total**: Todos os atributos e métodos das superclasses são herdados pelas subclasses, exceto se a resolução de conflitos de nomes excluir algum deles.

- **Compatibilidade de Domínios**: Os atributos herdados possuem o mesmo domínio ou um domínio especializado em relação aos atributos definidos na superclasse.

Basicamente três tipos de alterações podem ser realizadas sobre o grafo de herança entre classes: no conteúdo de um nodo, em um arco ou em um nodo. A lista de operações permite 27 diferentes alterações. A semântica de algumas delas pode ser encontrada em [KIM 89].

A principal qualidade desta proposta é definir uma lista de alterações bem mais ampla que a do GemStone, tendo explorado muitas das alternativas possíveis de modificação de um esquema conceitual. Estabelecer regras claras para selecionar um dos atributos possíveis quando existir conflito de herança é importante. Isto torna a semântica das operações mais complexa, porém resulta em maior facilidade de uso.

Apesar do modelo de dados suportar objetos versionados, não está claro se versões podem ter estruturas diferentes, resultantes de operações de evolução de esquemas.

2.5 SMM

O Schema Manipulation Mechanism (SMM) foi desenvolvido para a área de Engenharia de Software, visando a facilitar a reusabilidade e a modularização de programas no processo de desenvolvimento de sistemas de grande porte [RUG 91]. O SMM considera um esquema como sendo composto por dados e operações que podem ser reutilizadas em outro contexto.

O modelo de dados utilizado, chamado Modelo E/D [MOT 90], trata as características estruturais e comportamentais de forma integrada. Um objeto é um elemento do mundo real distintamente identificado devendo pertencer a uma ou mais classes. Toda classe deve estar associada a um tipo. Relacionamentos de generalização podem ser definidos entre tipos. Tipo pode ser **atômico** (Integer, Real, String, Bool ou Und), **pré-definido** (UserType, TypeConstraint, Entity, Relation, Action, Rule, Transaction ou MetaRule) ou definido pelo usuário utilizando um dos construtores disponíveis (Set, List, Record). Herança múltipla de propriedades é permitida e conflitos de nomes devem ser resolvidos pelo usuário escolhendo entre uma das possibilidades: redefinir o atributo na subclasse ou selecionar um entre os possíveis nas superclasses.

2.5.1 Evolução de Esquemas

O objetivo da proposta de evolução de esquemas é permitir combinar esquemas já existentes obtendo um novo esquema. A geração do novo esquema é controlada através do uso de invariantes que definem a configuração legal de um esquema. Os invariantes definidos são as seguintes:

- **Herança Estrita**: A estrutura definida para o supertipo é herdada para seus subtipos, exceto quando houver conflitos de nomes. Para o caso de métodos, diferenças nas suas assinaturas são consideradas.

- **Correspondência de Estrutura**: A estrutura dos objetos corresponde à estrutura declarada para o tipo.
- **Nomes de Tipos Únicos**: Os nomes dos tipos declarados devem ser únicos no esquema.
- **Manutenção da Hierarquia de Generalização**: Sejam A e B tipos, $A < B$ na hierarquia de generalização, isto é, A é subtipo de B, L_{AB} o número de classes entre A e B na hierarquia. Após combinar esquemas, no novo esquema a relação $A < B$ deve permanecer válida, mesmo que L_{AB} seja alterada.

Há dois modos de operação do sistema: **automático**, sem supervisão do projetista, com o sistema controlando os invariantes, e **assistente**, sobre controle e responsabilidade do projetista, que pode violá-los. Há quatro operações possíveis entre esquemas:

- União
- Extensão
- Diferença
- Alteração de nome de tipo

Destas, somente a quarta operação envolve apenas um esquema.

A propagação das alterações para as instâncias é realizada através de um processo de inicialização que gera **objetos exceção** com os mesmos identificadores. Quando estes forem referidos, este processo intercepta o acesso e gera um novo objeto modificado com um identificador diferente. Outras referências ao objeto antigo, quando utilizadas, serão modificadas para o objeto novo.

Um mecanismo de erros é proposto para resolver incorreções nos métodos do tipo. Para cada tipo, uma rotina de tratamento para situações de erro gerada por modificações no esquema pode ser definida pelo usuário para solucionar o problema.

Este trabalho propõe soluções para a evolução de esquemas diferentes das demais, pois baseia-se em operações de combinação de dois esquemas já existentes para obter um terceiro. Isto facilita a reutilização de definições de bases de dados em operação, incrementando o processo de desenvolvimento de sistemas. Por outro lado, obriga o projetista a sempre escrever um segundo esquema para realizar operações simples, como incluir um novo atributo, quando o natural seria permitir sua realização sobre o próprio esquema a ser alterado.

2.6 O_2

O_2 é um sistema de gerência de banco de dados orientado a objetos (SGBDOO) atualmente comercializado pela O_2 Technology [LEC 88, O_2T 91, DEU 91]. No O_2 a informação está organizada em objetos, que incorporam através do modelo de dados, baseado em conceitos de abstração, parte da semântica dos dados e através de métodos de acesso o seu comportamento.

Uma classe pode possuir uma ou mais subclasses, que herdam suas propriedades através do processo de herança. Herança múltipla é permitida. A resolução de conflitos de nomes deve ser feita pelo projetista. Uma propriedade de uma classe pode ser definida por objetos de outra classe, de acordo com o conceito de agregação. Esta composição pode ser através de conjunto, lista ou um atributo. Uma classe pré-definida, chamada *Object*, é a superclasse "default" das demais.

Métodos podem ser privados, visíveis apenas para a classe, ou públicos, visíveis para todo sistema. Estes métodos são implementados em uma linguagem específica chamada O_2C. Uma biblioteca de métodos para a classe *Object* acompanha o sistema. Estes métodos são herdados por todas as classes definidas pelo usuário.

Cada objeto no sistema possui um identificador único definido na sua criação. A classe a que pertence um objeto deve ser indicada neste momento. Um mesmo objeto pode fazer parte de diversos outros. Deste modo, caso o valor de um

dos seus atributos seja alterado, todos objetos compostos que o incluem perceberão a modificação realizada.

2.6.1 Evolução de esquemas

O mecanismo de evolução de esquema implementado para o O_2 [ZIC 91, O_2T 91] define, inicialmente, dois tipos de consistência para esquemas conceituais:

- **Consistência Estrutural**: Refere-se à parte estática do esquema. Este tipo de consistência é mantido por propostas já apresentadas (GemStone, Orion). Um esquema está estruturalmente consistente se a estrutura de classes for um grafo acíclico direcionado, o escopo de todos os atributos e métodos estiver correto, não houver conflitos de nome e houver compatibilidade entre valores armazenados e domínios dos atributos.

- **Consistência Comportamental**: Refere-se à parte dinâmica da base de dados. Um esquema está consistente se todos os métodos respeitam a sua assinatura e se durante a execução dos métodos não ocorrer erros ou resultar valores inesperados. Para cada método definido no esquema, uma lista de ocorrências de atributos e outros métodos referidos é mantida. Isto permite, por exemplo, que ao se remover um atributo de uma classe, sejam informados ao projetista todos os métodos que tornaram-se inválidos por utilizarem o atributo agora não existente. Cabe então ao projetista realizar as correções necessárias.

A lista de operações possíveis para evolução de esquemas é muito semelhante à existente para o ORION, apenas a semântica é diferente entre ambas, pois no ORION não há preocupação com consistência comportamental. Esta proposta é inovadora por ser a única a preocupar-se com a consistência comportamental do esquema conceitual. Embora seja difícil mantê-la, o mecanismo apresentado consegue indicar ao projetista quais são os métodos afetados pela alteração realizada, sendo

responsabilidade do projetista adequá-los ao novo esquema. Quanto à consistência estrutural o trabalho não difere substancialmente de outros já apresentados.

2.7 Conclusão

Neste capítulo, as principais propostas de evolução de esquemas existentes para bancos de dados orientados a objetos foram descritas, comparadas e criticadas. No capítulo seguinte, uma proposta de evolução de esquemas para um modelo de dados genérico é apresentada, procurando aproveitar as qualidades de todas anteriormente estudadas, de uma forma coerente, com o objetivo de facilitar a tarefa do projetista de software.

3 O MODELO DE DADOS

Neste capítulo, o modelo de dados no qual se baseará o mecanismo de evolução de esquemas é apresentado em suas principais características. Este modelo possui as principais características de orientação a objetos, conforme definidas no Manifesto de Atkinson et al. [ATK 89], sendo muito semelhante ao de diversos sistemas já existentes de bancos de dados e linguagens orientadas a objetos [BUT 91, MAT 91, DAV 86, LAL 90, BRY 93, WIL 90, LAM 91], notadamente o do O_2 [DEU 91] e ORION [KIM 89].

Acrescentou-se ao modelo de dados um plano de versões, que é desenvolvido e apresentado. Este plano de versões será utilizado como forma de registro das etapas do desenvolvimento do esquema conceitual. Orientação a objetos e versões são largamente utilizadas em aplicações não-convencionais como forma de acelerar o desenvolvimento dos sistemas e obter melhores resultados na resolução de problemas.

3.1 Objetos

Objetos são uma abstração de uma entidade existente no mundo real, representada por um estado, descrito através de atributos, e um comportamento, descrito através de métodos. Atributos possuem um domínio, que define o conjunto de valores permissíveis para o atributo. Alguns domínios são pré-definidos no sistema, como inteiros, caracteres, reais e booleanos. Métodos descrevem o comportamento dos objetos e somente através deles é permitida a sua manipulação. Cada método possui uma assinatura própria, formada por seu nome, uma lista de parâmetros e tipo de retorno. Objetos com características semelhantes são agrupados em classes.

Objetos complexos são construídos a partir de outros objetos mais simples através de construtores [ATK 89]. Assim, torna-se possível descrever a composição de um objeto, chamado agregado, em termos de outros objetos, os componentes, para existir consistentemente [SMI 77]. Todas as classes e tipos definidos no es-

quema também são domínios válidos na construção de um objeto agregado. Três construtores estão definidos:

- *Set*: Conjuntos podem ser utilizados para associar um número variável de objetos da mesma classe, chamados elementos [HAM 81].
- *Lista*: Listas podem ser utilizadas para associar um número limitado de objetos de uma mesma classe ordenadamente.
- *Tuple*: Tuplas podem ser utilizadas para associar objetos de diferentes classes em uma única classe.

Estes construtores podem ser aplicados ortogonalmente. Assim, por exemplo, é possível definir tuplas de vetores, conjuntos de tuplas, vetores de conjuntos de vetores ou qualquer outra combinação que o projetista entender necessária.

A ligação entre os objetos é realizada através da definição de atributos que possuem como domínio outras classes pertencentes ao esquema, diretamente ou através de um destes construtores.

3.2 Tipos e Classes

Os conceitos de classe e tipo são suportados neste modelo, embora tenham definições diferentes, seguindo a proposta de [BEE 90]. Tipos são entendidos como descritores de características estruturais que serão incorporadas a uma ou mais classes, sendo definidos como domínios de atributos. Podem ser utilizados para evitar a duplicação de sua descrição se mais de uma classe possuir objetos do mesmo tipo. Assim, é possível definir o tipo *Data*, como composto por três campos diferentes, *Dia*, *Mês* e *Ano*. Esta definição pode ser reaproveitada em várias classes para armazenar atributos do tipo *Data*. No entanto, agindo assim, nenhuma operação ou instância é definida para ela.

Classes são vistas como descritores das características comportamentais básicas das instâncias que a ela pertencem. São repositórios de objetos com as mesmas propriedades, compartilhando a mesma definição de atributos e métodos. Todas as instâncias existentes na base de dados devem pertencer a uma determinada classe definida no esquema conceitual. Deste modo, objetos idênticos podem ser organizados em um mesmo repositório, para o qual um conjunto de operações aplicáveis a todos está definido.

Segundo Beeri [BEE 90], tipos são utilizados para denotar a estrutura e a extensão de seus elementos. Extensão é o conjunto de todos os possíveis valores que pertencem ao(s) domínio(s) definido(s) para o tipo. A estrutura de um tipo é fixa e a sua extensão definida pela sua estrutura, portanto, a extensão de um tipo é fixa. Classes também possuem uma estrutura, porém sua extensão é definida pelo usuário, que associa e retira objetos de uma determinada classe.

3.3 Subclasses e Subtipos

Um mecanismo de herança de atributos e métodos definidos em classes está presente no modelo. Ele possui, basicamente, duas vantagens: é uma ferramenta de modelagem poderosa, porque possibilita uma descrição da realidade modelada de forma precisa e concisa, e permite o compartilhamento de especificações desta realidade [ATK 89].

No entanto, é necessário distinguir entre subclasses e subtipos, pois estes conceitos estão freqüentemente confundidos na literatura existente. Uma subclasse *SB* possui os mesmos atributos e métodos das suas superclasses *SP* e pode ter zero, um ou mais atributos e métodos definidos localmente, além de redefinir alguns atributos herdados. Note-se que por esta definição, atributos herdados pelas subclasses podem ser redefinidos livremente nos níveis inferiores, pois nenhuma restrição está imposta na definição acima.

Visando prover também o mecanismo de subtipos, uma limitação é imposta à redefinição de atributos herdados. Um atributo pode ser redefinido na subclasse se seu domínio for uma especialização do domínio definido para o atributo na superclasse, ou seja, se o domínio do atributo da subclasse for subconjunto do domínio do atributo na superclasse. Métodos podem ser redefinidos se cada um dos domínios da sua assinatura estiver contido no respectivo domínio definido na superclasse.

Define-se que uma classe C' é uma subclasse de outra C, notada $C' \sqsubseteq C$, se todos os objetos pertencentes a C' pertencem também a C. Ou, em outros termos, C' é uma subclasse de C se para qualquer objeto que satisfaça a descrição da classe C' satisfizer também a descrição de C [AME 90].

3.4 Herança Múltipla

Embora seja considerada como uma característica opcional pelos autores do manifesto de Atkinson et al. [ATK 89], este modelo, a exemplo de outros sistemas [DEU 91, BAN 87, MAT 91], permite que uma subclasse possua uma ou mais superclasses, criando um processo de herança múltipla através de um grafo de relacionamentos de generalização. Para manter o grafo conexo e prover alguns serviços comuns a todas as classes, como criação e remoção de objetos, por exemplo, uma superclasse comum a todas está definida. Esta classe é chamada *GLOBAL* .

Dois conjuntos de classes relacionadas pelo conceito de generalização podem ser considerados para cada classe existente:

- O conjunto das superclasses de uma classe C, escrito como $SP(C)$. Este conjunto nunca pode ser vazio, pois toda classe está obrigatoriamente ligada, no mínimo, à classe genérica *GLOBAL* .

- O conjunto das subclasses de uma classe C, escrito como $SB(C)$. Este conjunto pode ser vazio, pois uma classe não é obrigada a possuir subclasses.

Se $C \in SP(C')$ então $C' \in SB(C)$, para quaisquer classes C e C'.

No processo de herança múltipla, conflitos de nomes podem ocorrer. Acontecem quando uma determinada subclasse possui em duas ou mais superclasses um atributo/método definido com o mesmo nome, mas domínios diferentes. Por exemplo, a classe *Barcos* pode ser definida como subtipo das classes *Veículos_Motorizados* e *Veículos_Aquáticos*. Na classe *Veículos_Motorizados* pode haver uma propriedade chamada *Velocidade_Máxima* expressa em quilômetros por hora, enquanto na classe *Veículos_Aquáticos* pode haver uma outra propriedade, também chamada *Velocidade_Máxima*, mas expressa em nós. Logo, duas propriedades com nomes iguais, mas domínios diferentes, serão herdadas pela classe *Barcos*.

Para resolver conflitos de nomes, algumas regras, por ordem de prioridade, estão definidas para o modelo de dados, semelhantes às existentes no ORION [BAN 87]:

1 - O projetista especifica claramente a resolução do conflito.

2 - O atributo/método é herdado do nível superior mais próximo onde está definido.

3 - Entre superclasses de um mesmo nível, uma ordem de prioridade é criada de acordo com o momento de criação da subclasse e associação de novas superclasses a ela. Neste caso, o atributo/método é herdado da primeira superclasse da lista.

A resolução de conflitos de nomes, no entanto, pode romper com o conceito de subtipos. Por exemplo, na figura 3.1 temos as classes C_1 e C_2. Ambas possuem atributos chamados A e B, porém com domínios diferentes. A subclasse C_3 herda as características de ambas e resolve o conflito de nomes existente optando

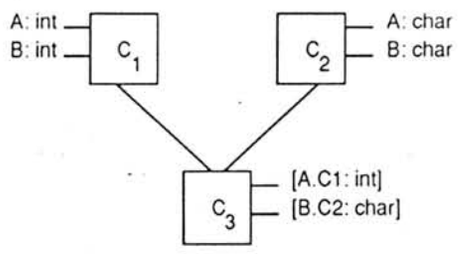

Figura 3.1: Exemplo de uma quebra na regra de subtipos

por $A.C_1$ e $B.C_2$. Esta opção está correta, porém instâncias da classe C_3 não satisfarão completamente nem a descrição da classe C_1 nem da classe C_2, ou seja, o tipo de C_3 não é um subtipo de suas superclasses. Portanto, existindo herança múltipla, nem todas as subclasses serão subtipos de suas superclasses [COO 90].

3.5 Encapsulamento

O conceito de encapsulamento permite ao projetista do sistema esconder dos usuários a implementação da classe e das operações que atuam sobre ela [ATK 89] e definir o comportamento lógico das instâncias pertencentes a classe. O usuário tem conhecimento somente da interface da classe, definida através de um conjunto de métodos que realizam alterações sobre instâncias de determinada classe. O encapsulamento provê, assim, uma forma de independência lógica dos dados, pois torna-se possível alterar a implementação da classe sem alterar a sua interface e, deste modo, preservar os programas que utilizam métodos da classe contra efeitos colaterais negativos, pois apenas os atributos do objeto referido na mensagem poderão ser afetados através deste mecanismo [AME 90].

Neste modelo, um método pode ser definido como uma tupla (N, P, D, F), onde N é o nome de identificação do método, P a lista de parâmetros p_i do método, D é o domínio do valor de retorno do método e F, o código fonte. O domínio D poderá ser o conjunto unitário $VOID$, que possui o valor especial *void*. Este valor especial permite que o método atue como um procedimento e não como função. Para cada parâmetro p_i pertencente a P, está associado um determinado

domínio pd_i. A cardinalidade da lista de parâmetros de um método é definida como sendo $\| P \|$. Todo método deve estar ligado diretamente a uma classe do esquema. A notação $M \twoheadrightarrow C$ indica que o método M está definido para a classe C.

Do mesmo modo como se permite a redefinição de atributos, em uma subclasse é possível redefinir um método herdado. Seguindo a definição de subclasse vista acima, um método $M'(N', P', D', F')$ redefinirá corretamente outro método $M(N, P, D, F)$ se e somente se:

1. $M \twoheadrightarrow C, M' \twoheadrightarrow C' \wedge C' \sqsubseteq C$.

2. $N' = N$.

3. $D' \subseteq D$.

4. $\| P' \| = \| P \| \wedge \forall i, 1 \leq i \leq \| P' \|, pd'_i \subseteq pd_i$.

Caso um método M' atenda a todas as condições impostas acima e redefina corretamente outro método M, herdado de uma ou mais superclasses, diremos que M' redefine M, representado como $M' \prec M$. Estas condições são semelhantes às existentes no sistema O_2 [DEU 91].

Esta restrição, chamada por Wegner [WEG 87] de subtipo horizontal, visa garantir que a operação associada a uma subclasse possui a mesma definição de interface, ou seja, que a aplicação poderá enviar uma mensagem a instâncias da classe ou de suas subclasses utilizando os mesmos parâmetros, mesmo que o método esteja redefinido em alguma subclasse. Além disso, torna possível definir um procedimento geral, que pode ser associado a diferentes classes de objetos, aumentando a reusabilidade de código e facilitando o desenvolvimento de sistemas complexos.

Porém, se através de resolução de conflitos de nomes em um processo de herança múltipla, a subclasse não se mantiver como subtipo de sua superclasse, não necessariamente suas instâncias responderão corretamente a métodos definidos no nível superior. No caso visto na figura 3.1, se houver um método M_1 na classe C_1 que utilize o atributo A para somar a um valor, uma instância da subclasse

C_3 conseguirá utilizá-lo corretamente. Porém, para outro método M_2, que realize cálculos sobre o atributo B, instâncias de C_3 não responderão corretamente, pois resolveram o conflito escolhendo o atributo B como definido na classe C_2, com o domínio *Char*.

3.6 O Plano de Versões

Versões são utilizadas como forma de armazenar diferentes estados de um mesmo objeto, tornando possível alcançar estados anteriores do objeto naturalmente [DIT 87, KAT 90]. Isto permite ao banco de dados reter informações sobre o aspecto evolucionário das informações armazenadas, mantendo um histórico completo de todas as etapas dos objetos na base de dados.

Considerando que o projeto de um esquema de objetos é a nossa aplicação, os objetos de interesse desta aplicação são instâncias, classes e métodos. Como evolução de esquema implica possíveis modificações em de todos estes objetos, e é nosso interesse manter o histórico de desenvolvimento em seus diferentes estados, instâncias, classes e métodos são objetos versionáveis. O uso de versões para todos componentes de um esquema também permite preservar a transparência de alteração de esquemas, como proposta na seção 4.1.

As versões de um objeto estão organizadas em um grafo acíclico e conexo. Apenas a versão inicial de cada grafo não possuirá uma antecessora. Para cada grafo de derivação de versões existe uma determinada versão corrente, considerada como sendo válida na configuração atual do esquema, sobre a qual serão realizadas todas as operações. Esta versão corrente, por "default" será a mais recente, porém o usuário poderá selecionar outra para ser a corrente.

As versões de um objeto podem ser classificadas de acordo com seu estado de desenvolvimento e consistência. Assim, versões de classes, métodos ou instâncias podem ser consideradas como **estáveis** ou **em trabalho** [DIT 87, LAM 91, KIM 90]. A estabilização de uma versão pode ser feita explicitamente pelo

usuário, ou implicitamente pelo sistema. Uma versão será promovida a estável, pelo sistema, quando:

- Possuir versões sucessoras.
- Possuir uma subclasse com uma versão estabilizada.
- Possuir uma versão de instância estabilizada.

Versões em trabalho podem ser alteradas, enquanto versões estáveis não. Alterações em versões estáveis só podem ser realizadas gerando uma nova versão. Nas versões em trabalho, podem ser realizadas a qualquer instante, inclusive alterações de esquema. Neste caso, o usuário poderá definir a criação de uma nova versão, estabilizando a atual.

3.6.1 Geração de Versões

Versões sempre podem ser criadas explicitamente pelo usuário para qualquer objeto na base de dados. Nas seguintes situações o sistema gera uma nova versão do objeto, copiando todas as características não alteradas:

- Para classes: modificação em um atributo, método associado ou se a versão de classe estiver estabilizada e para uma da suas superclasses for criada uma nova versão. Atributos podem ter seu nome ou domínio alterado e métodos podem ser acrescentados ou removidos.
- Para instâncias: modificações em atributos definidos na classe à qual a instância pertence. Cada versão de instância pertence a exatamente uma versão de classe e quando uma nova versão de classe é criada, por ter sido modificada, uma nova versão de todas as instâncias existentes naquele dado instante é gerada.
- Para métodos: alterações na assinatura do método, que podem ser nos seus parâmetros, domínios dos parâmetros e nome.

Alterações no esquema que não modificam a descrição da classe, por exemplo, a inserção de uma nova subclasse, não geram uma nova versão da classe nem de suas instâncias. Se a subclasse estiver em trabalho e for gerada uma versão de uma de suas superclasses, a versão de subclasse passa a herdar a descrição da nova versão da superclasse. O sistema também não gera novas versões se a semântica de um método ou os valores dos atributos forem alterados, se a versão estiver em trabalho, por não manter controle sobre eles. Nestes casos, o usuário é responsável por criá-las, se achar conveniente.

Estas regras visam a manter a consistência da base de dados e permitir um mecanismo flexível e eficiente para a evolução de esquema, sem haver uma geração desenfreada de novas versões de objetos, o que saturaria o sistema e dificultaria sua utilização. O armazenamento de diferentes estágios do esquema em versões permite ao projetista retornar a um estado anterior no desenvolvimento do esquema/aplicação, facilitando este processo, pois permite a definição de novas alternativas à implementação já existente e a redefinição de soluções adotadas erroneamente.

3.6.2 Conceitos de Abstração e Versionamento

Os conceitos de abstração, como descritos anteriormente, são adaptados para o modelo de versões. Como a agregação é mantida entre classes diferentes através da definição de atributos que possuem como domínio outra classe, nenhuma versão desta classe é especificada. Cada instância da classe agregado possui o atributo cujo valor será um identificador de versão de uma instância da classe definida como componente. Nenhuma restrição é realizada quanto à versão deste componente nem quanto à sua versão de classe, permanecendo a aplicação responsável por definir, se houver necessidade, algum controle mais específico. Do ponto de vista do sistema, é possível até que uma única instância de agregado seja composta por diferentes versões de um mesmo componente.

Um mecanismo auxiliar de notificação de mudanças é acionado sempre que uma versão de componente for alterada ou a partir dela for derivada uma nova versão. Nesta situação, todas as versões de agregados do qual participa são notificadas através de um indicador ("flag") que um componente seu foi alterado ou possui uma versão mais nova. O projetista, ao utilizar o objeto agregado, receberá esta informação e poderá escolher entre manter a versão anterior ou substitui-la. Caso modifique a versão do agregado, e, este seja componente de outros objetos, neste momento estes serão notificados. Este mecanismo de notificação é semelhante ao proposto para o sistema ORION [CHO 88].

Na generalização, o processo de herança de propriedades é realizado sempre através das versões mais novas pertencentes a hierarquia, assegurando a permanente atualização dos níveis inferiores, pois se uma nova versão da superclasse foi criada, novas versões das subclasses também serão, herdando sempre a descrição mais recente de sua superclasse.

Para generalização, o mecanismo de notificação não é necessário, pois a modificação em uma subclasse não altera a superclasse, além disso, não há ligações entre instâncias de níveis diferentes na hierarquia. E, no sentido contrário, se uma versão de superclasse for modificada ou derivar uma nova versão, por herança, as versões de subclasses serão automaticamente modificadas ou gerarão uma nova versão.

3.6.3 Exemplo de Evolução de Esquema com Versionamento

A figura 3.2 mostra um exemplo de evolução de esquema com versionamento. Classes são representadas por retângulos, instâncias por círculos, relacionamentos de generalização e instanciação por linhas contínuas. A versão corrente está com a borda hachurada. Setas simples indicam uma derivação determinada pelo usuário e duplas derivações geradas pelo sistema. A figura 3.2a mostra a definição

inicial feita pelo projetista, depois de uma instância I_1 ter sido associada à classe C_2.

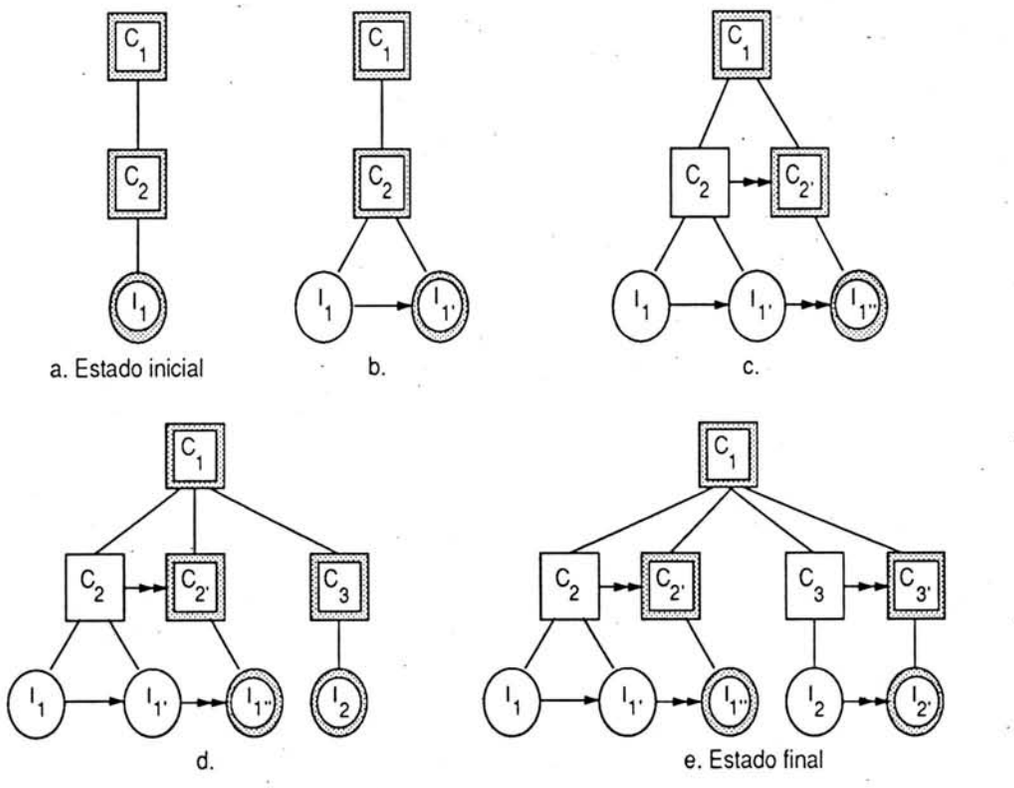

Figura 3.2: Exemplo de evolução de esquemas e versionamento

No passo b, o usuário gerou uma nova versão da instância I_1, através de uma ou mais alterações em valores dos seus atributos.

No passo c, o projetista alterou a descrição da classe C_2. O sistema automaticamente gerou uma nova versão desta classe (e de suas subclasses, se existissem, porque a descrição destas também teria sido alterada). Na subclasse C_2, como já existia uma instância, uma nova versão desta foi gerada, seguindo a descrição atual.

No passo d, uma nova subclasse, C_3, foi incluída para a superclasse C_1. Como esta operação não altera a descrição das instâncias de C_1, não é gerada uma nova versão desta. Uma instância foi colocada em C_3.

No passo e, a subclasse C_3 teve sua descrição alterada por uma operação qualquer. O sistema gerou uma nova versão $C_{3'}$ para a classe e sua instância, $I_{2'}$.

3.6.4 Contexto de Esquema

Contextos de esquema são conjuntos de versões de classes, instâncias e métodos que se relacionam corretamente de acordo com as descrições existentes em cada uma delas.

O objetivo de um contexto é definir um conjunto composto pelas versões de objetos que se relacionam coerentemente, isto é, permitir a seleção da versão de método adequada, para a versão de instância que recebeu a mensagem, pertencentes a uma específica versão de classe. Portanto, todas as referências realizadas dentro de um contexto devem estar corretas, pois contextos são utilizados para assegurar a coerência na utilização de métodos, classes e instâncias, evitando interrupções na execução de um método ou operação de alteração de esquema.

Por exemplo, na remoção de um atributo de uma classe, o sistema deve realizar as seguintes tarefas:

- Uma nova versão de classe deve ser criada, removendo o atributo.
- A remoção deve ser propagada para as subclasses.
- Novas versões de instâncias devem ser criadas, sem o respectivo atributo.
- O usuário deve ser notificado se existem métodos que utilizavam o atributo. Se existir algum, o usuário deverá providenciar novas versões destes métodos.

Estas novas versões de métodos só deverão ser aplicadas sobre a nova versão da classe e das instâncias. Além disso, a uma versão de classe poderão estar associadas várias versões de uma mesma instância ou método. Quando uma mensagem é enviada para um objeto, o sistema deve identificar a versão de instância (a corrente se o usuário não especificar outra), a versão de classe a que a instância está associada e a versão correta do método a ser aplicado. E se for uma alteração de esquema, incluir também versões de todos os objetos que podem ser afetados pela

operação, com o objetivo de manter a coerência entre todos os objetos envolvidos, para os quais, eventualmente, poderão ser geradas novas versões.

Assim, um contexto de uma versão de objeto $CTX(OBJ_n)$, é definido como sendo a união de:

1. A versão de objeto, OBJ_n, para a qual se está definindo o contexto.

2. A versão m da classe a qual a versão de objeto está ligada, se não for uma versão de classe.

3. A última versão das superclasses à qual a versão da classe definida no item 2 está ligada, até o nível da classe pré-definida *GLOBAL* .

4. A última versão de todas as subclasses, até o nível mais inferior, da versão de classe definida no item 2.

5. As últimas versões de instâncias e métodos das versões de classes que pertencem ao contexto.

Quando uma mensagem é enviada a um objeto, o sistema calcula o contexto de esquema deste objeto e a versão de método escolhida é a mais recente no mesmo contexto do objeto. Os componentes 1, 2, 3 e 5 permitem ao sistema realizar esta operação. O componente 4 é necessário para propagar mudanças, realizadas em uma superclasse, para todas as suas subclasses. Por exemplo, a remoção do atributo na superclasse resulta na remoção deste atributo em todas as suas subclasses.

Desde modo, dentro de um esquema convivem vários contextos diferentes, de forma semelhante à definição de configuração existente em [CEL 91a]. Há apenas um **contexto corrente** em determinado instante de tempo. Caso o usuário escolha uma versão de um objeto qualquer existente na base de dados, o contexto corrente será recalculado, mesmo que o objeto escolhido já pertença ao contexto corrente, porque o seu contexto específico pode formar um conjunto diferente do contexto corrente, embora, provavelmente, a interseção de ambos resulte um conjunto próximo a ambos.

A troca de versão corrente em um nível pode resultar em troca de versão corrente em todos os níveis, pois o contexto deve ser recalculado. Deste modo, a base de dados retorna ao estado anterior mais recente onde a nova versão corrente era válida e assegura um contexto de esquema coerente com a versão selecionada. O envio de uma mensagem a determinada versão de instância aciona a versão adequada do método pedido. Esta versão de método atuará no contexto válido na sua definição e para o qual está correta. Caso não exista uma versão adequada do método, a mensagem falhará. Se durante a execução de um método, este enviar uma segunda mensagem para outro objeto, provavelmente haverá uma troca de contexto para permitir a execução correta deste segundo método. Quando este terminar, o contexto retorna para o contexto do método original, que enviou a segunda mensagem.

Um contexto corrente também é alterado caso uma instância ou método pertencente a ele seja versionado. Então, a versão existente deixa de pertencer ao contexto, sendo a nova versão acrescentada a ele. Caso uma classe seja versionada, com o conseqüente versionamento das instâncias e, eventualmente, das subclasses, um novo contexto é criado, contendo as novas versões da classe e das suas instâncias e subclasses.

No caso de exemplo mostrado na figura 3.3, a tabela 3.1 mostra os contextos de todos os objetos envolvidos.

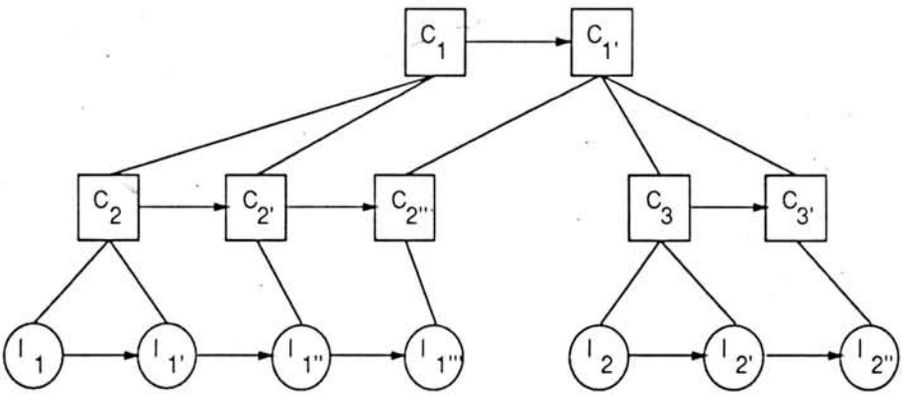

Figura 3.3: Exemplo de um esquema com versões.

Tabela 3.1: Tabela de contextos de objetos

$CTX(C_1) = \{C_1, C_{2'}, I_{1''}\}$	$CTX(I_1) = \{C_1, C_2, I_1\}$
$CTX(C_{1'}) = \{C_{1'}, C_{2''}, I_{1'''}, C_{3'}, I_{2''}\}$	$CTX(I_{1'}) = \{C_1, C_2, I_{1'}\}$
$CTX(C_2) = \{C_2, C_1, I_{1'}\}$	$CTX(I_{1''}) = \{C_1, C_{2'}, I_{1''}\}$
$CTX(C_{2'}) = \{C_{2'}, C_1, I_{1''}\}$	$CTX(I_{1'''}) = \{C_{1'}, C_{2''}, I_{1'''}\}$
$CTX(C_{2''}) = \{C_{2''}, C_{1'}, I_{1'''}\}$	$CTX(I_2) = \{C_{1'}, C_3, I_2\}$
$CTX(C_3) = \{C_3, C_{1'}, I_{2'}\}$	$CTX(I_{2'}) = \{C_{1'}, C_3, I_{2'}\}$
$CTX(C_{3'}) = \{C_{3'}, C_{1'}, I_{2''}\}$	$CTX(I_{2''}) = \{C_{1'}, C_{3'}, I_{2''}\}$

3.7 Mecanismo de Mensagens

Um sistema orientado a objetos pode ser descrito como um conjunto de objetos que comunicam-se entre si para alcançar um determinado resultado [LAL 90]. Cada objeto pode ser visto como um pequeno computador, com um estado (definido pelos valores dos seus atributos) e um conjunto de operações (definido pelos métodos associados a ele). Uma operação é realizada através do envio de uma mensagem a um objeto, chamado **receptor**. Um **seletor** deverá identificar qual o método, entre os pertencentes à classe, que deverá ser ativado. Este mecanismo é tipicamente síncrono, isto é, um método só é ativado pelo seletor se o objeto estiver vago, ou seja, nenhum outro método estiver sendo executado para o mesmo receptor.

Neste modelo, o receptor é uma versão específica de um objeto, reconhecida pelo seu identificador único. Ao receber uma mensagem, a versão de objeto

é selecionada como corrente e todo o contexto é alterado, conforme a definição da seção anterior. O seletor atuará neste novo contexto, escolhendo a versão de classe a qual o receptor está ligado. Nesta versão de classe, a versão mais recente do método adequado será ativada. Porém, para esta versão de classe pode não estar definida uma versão do método pedido, ocasionando uma falha no envio da mensagem. Outra possibilidade de erro é a assinatura da versão de método escolhida não corresponder à lista de parâmetros e tipo de retorno utilizada pelo remetente da mensagem.

Assim, a partir da situação mostrada na figura 3.4, onde a classe C_1 possui quatro versões, uma instância I_1 com cinco versões e um método M_1 com quatro versões, caso seja enviada uma mensagem para uma versão desta instância ativando o método M, a tabela 3.2 mostra qual versão de método será ativada para cada uma das versões de instância possíveis.

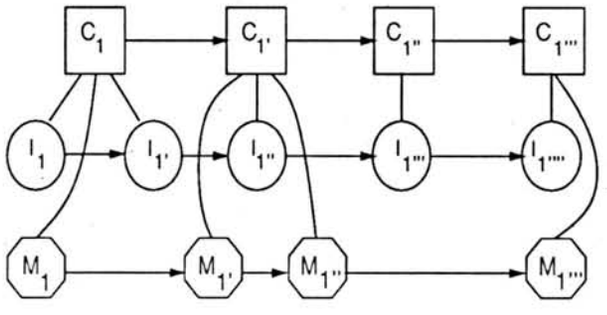

Figura 3.4: Versões de uma classe com instância e método associados

Tabela 3.2: Versão do método acionada através do envio de uma mensagem a uma versão de objeto

Versão do objeto receptor	Versão de método ativada
I_1	M_1
$I_{1'}$	M_1
$I_{1''}$	$M_{1''}$
$I_{1'''}$	ERRO
$I_{1''''}$	$M_{1'''}$

Para as versões de instância I_1 e $I_{1'}$, a versão de método M_1 é acionada por ser a única associada a versão de classe. Para a versão de objeto $I_{1''}$ a versão $M_{1''}$ é escolhida por ser a mais recente. Para a versão de objeto $I_{1'''}$ um erro ocorre porque

nenhuma versão do método está associada à mesma versão da classe, pois a versão anterior do método, após a alteração do esquema ficou inválida para a nova versão de classe. Neste momento, uma lista contendo todas as versões possivelmente inválidas foi passada para o projetista, que não providenciou uma nova versão do método compatível com esta versão da classe. Para a versão de objeto I_1^{m}, a versão de método M_1^{m} é escolhida por estar definida para a versão de classe. Caso estivéssemos considerando a assinatura de cada versão de método, outros erros poderiam ocorrer.

3.8 Outras características

Várias outras características de um banco de dados orientado a objetos são definidas no manifesto de Atkinson et al. [ATK 89] e estão presentes em diversos sistemas. Neste trabalho estão sendo consideradas apenas as características relacionadas ao modelo de dados. No entanto, para o mecanismo de evolução de esquemas algumas facilidades são importantes. A partir da descrição do modelo, o banco de dados deverá suportar os mecanismo de sobreposição ("overriding"), ligação postergada e sobrecarga ("overloading"). Estes mecanismos são importantes para aumentar a reusabilidade de software, essencial no paradigma de orientação a objetos.

O sistema deverá realizar a cada inserção ou alteração de um método uma análise sintática para identificar quais atributos e métodos estão sendo utilizados. Esta informação será posteriormente utilizada para verificações na evolução do esquema.

Alguns tipos básicos não são representados como objetos, por exemplo, inteiros, booleanos, reais e caracteres. Ainda assim, classes, tipos, métodos e instâncias podem ser vistos pelo usuário de maneira uniforme como sendo objetos versionáveis, embora possuam semânticas diferenciadas.

Todos os objetos na base de dados possuem um identificador único, definido pelo sistema, independente de valores armazenados em atributos das instâncias.

Cada versão de um mesmo objeto também podem ser identificada unicamente pelo sistema. Seu identificador é composto pelo identificador do objeto a que pertence mais um número de versão.

3.9 Conclusão

Neste capítulo foram apresentadas as principais características do modelo de dados a ser utilizado ao longo deste trabalho, sobre o qual o mecanismo de evolução de esquemas foi proposto. Trata-se de um modelo de dados orientado a objetos, que permite herança múltipla, agregação através de domínios de atributos, redefinição de atributos e métodos nas subclasses e provê um mecanismo de encapsulamento de objetos através de métodos. Um ambiente com versões de classes, métodos e instâncias para este modelo foi descrito e seu funcionamento explicado, com o objetivo de fornecer ao projetista da base de dados suficiente flexibilidade para a alteração do esquema.

4 O MECANISMO DE EVOLUÇÃO DE ESQUEMAS

Neste capítulo o mecanismo de evolução de esquemas é descrito. Este mecanismo baseia-se no conceito de invariantes, condições básicas para a base de dados ser considerada válida e consistente pelo sistema. Estes invariantes são definidos na primeira seção do capítulo. A seguir, as diversas operações que podem ser realizadas sobre um esquema são descritas, detalhando para cada uma as suas opções e conseqüências. Finalmente, alguns mecanismos auxiliares para aumentar a transparência de alteração de esquemas são esboçados.

4.1 Transparência de Alteração de Esquemas

Transparência de alteração de esquemas é uma extensão para o modelo orientado a objetos do conceito de independência lógica de dados: grau de imunidade das aplicações a mudanças na definição do esquema conceitual da base de dados. Ou seja, por mais que este esquema mude, a aplicação continuará funcionando da mesma forma. Esta característica é extremamente importante para tornar um mecanismo de alterações de esquema mais prático e útil, pois segundo Segal [SEG 93], quanto mais transparente a alteração realizada for para o sistema, mais as pessoas a utilizarão.

Uma adaptação inicial deste conceito, para a teoria de tipos abstratos de dados foi realizada por Zdonik [ZDO 90], considerando a existência de versões de tipos. Para Zdonik, transparência de alteração de tipos significa:

1 - Manter velhos programas funcionando com novas versões de um tipo ou de suas instâncias.

2 - Permitir que novos programas reconheçam versões antigas do tipo ou de suas instâncias.

Considerando que no paradigma de orientação a objetos o usuário reconhece apenas a definição dos métodos que se aplicam sobre uma classe, manter a transparência de alteração de esquemas significa:

As assinaturas e a semântica dos métodos de uma classe não devem ser alteradas, apenas a sua implementação.

Assim, poderíamos garantir facilmente que velhos e novos programas utilizariam os objetos corretamente, não importando, realmente, a estrutura interna dos métodos e da classe. Porém, muitas vezes o projetista poderá ter a necessidade de alterar inclusive a assinatura e a semântica de determinado método, refletindo uma mudança na especificação da classe.

Além disso, algumas modificações na estrutura da classe, embora não sejam vistas diretamente pelo mundo exterior, podem resultar, obrigatoriamente, em alteração do comportamento da classe. Por exemplo, suponhamos que a classe *Pessoa* possua o atributo *Data de Aniversário*. O método que recupera um objeto pertencente a esta classe, fa-lo-á incluindo na resposta a data de aniversário. Em um instante posterior, esta data de aniversário é retirada da estrutura da classe. O método de recuperação de um objeto não fornecerá mais a data de aniversário e o programa que acionou o método provavelmente não conseguirá realizar as operações relativas a esta data, como imprimi-la ou consultá-la para enviar um cartão de felicitação na véspera do dia de aniversário. Ou seja, a simples proibição de alterações da assinatura de métodos resultaria em um sistema excessivamente restrito, embora fosse uma maneira simples de garantir o princípio de transparência.

Neste aspecto, este trabalho representa um avanço a partir da idéia proposta por Zdonik [ZDO 90], pois preocupa-se não apenas em manter consistente o esquema, mas também em averiguar a forma como a parte interna dos objetos pode ser modificada sem trazer alterações importantes para o mundo ao seu redor.

4.2 Invariantes de esquema

Invariantes de esquema são condições básicas que sempre devem ser satisfeitas para que o estado do esquema seja válido. Estes invariantes são utilizados em vários outros sistemas [BAN 87a, PEN 87, RUG 91, ZIC 91] exatamente para verificar a consistência da alteração realizada. Após a realização de uma evolução no esquema, os invariantes são verificados para assegurar um novo estado correto da base de dados. Caso o projetista tenha violado um destes invariantes, a operação é desfeita.

Neste mecanismo, dois conjuntos de invariantes são propostos, estruturais e comportamentais. Algumas alterações no esquema validadas pelos invariantes estruturais podem resultar em alteração nos valores de atributos armazenados. Como estes valores são reconhecidos pelo sistema, este pode realizar as modificações necessárias para mantê-los coerente com o novo estado do esquema. Já quanto às modificações verificadas pelos invariantes comportamentais, aquelas realizadas sobre os métodos, o sistema valida apenas o relacionamento existente entre eles e as classes do esquema. Nada é verificado quanto à utilização que as aplicações fazem destes métodos. Assim, algumas modificações consideradas corretas pelo sistema podem ocasionar problemas quando os métodos forem acionados pelas aplicações. O sistema passa para o projetista uma lista de métodos afetados por uma modificação do esquema, sendo responsabilidade do projetista verificar como estes métodos estão sendo acionados pelas aplicações e se realmente ocasionarão problemas em tempo de execução.

4.2.1 Invariantes Estruturais

Os invariantes estruturais visam assegurar a integridade do conjunto de classes e instâncias com suas descrições internas e relacionamentos de generalização e agregação. Foram divididos em dois subconjuntos, um relativo às classes e outro às instâncias armazenadas:

4.2.1.1 Invariantes Estruturais de Classe

1 - Os relacionamentos de generalização resultam em um grafo acíclico e conexo, com todas as classes ligando-se, no mínimo, à superclasse *GLOBAL* , pré-definida no esquema.

2 - Todas as classes possuem nomes únicos.

3 - Todos os atributos existentes possuem nomes únicos, em relação à classe a que pertencem. Conflitos de herança devem estar resolvidos.

4 - Todos os atributos pertencentes ao conjunto de superclasses são herdados, exceto se houver redefinição do atributo na subclasse ou se a resolução de conflito de nomes excluir algum atributo.

5 - Todos os atributos herdados só podem estar redefinidos para domínios que sejam subconjuntos dos domínios de suas definições na superclasse.

6 - Todas as classes referidas como domínios de atributos existem.

4.2.1.2 Invariantes Estruturais de Instâncias

1 - Todos os objetos referidos estão presentes na base de dados.

2 - O valor definido para um atributo de uma versão de instância pertence ao domínio definido para a versão de classe ao qual está associado. O valor NULL pertence a todos os domínios, podendo ocorrer em qualquer atributo.

4.2.2 Invariantes Comportamentais

Os invariantes comportamentais asseguram a integridade do conjunto de métodos declarados no esquema.

1 - Todos os métodos existentes em uma classe possuem nomes únicos. Conflitos de herança devem estar resolvidos.

2 - Todos os métodos pertencentes ao conjunto de superclasses são herdados, exceto se houver redefinição do método na subclasse ou se a resolução de conflito de nomes excluir algum método.

3 - Todos os métodos herdados só podem ser redefinidos para comportamentos mais específicos ou idênticos ao de sua definição na superclasse.

Na verificação dos invariantes, considera-se um domínio D, definido sobre uma classe C, como sendo um subconjunto de outro domínio D', definido sobre uma classe C', se e somente se, $C \sqsubseteq C'$, direta ou indiretamente. Se a classe C não possuir nenhuma relação de generalização com a classe C', então é dito que $D \neq D'$. Por exemplo, a classe *Carros* é subclasse de *InventosHumanos*, ou seja, $D(Carros) \subset D(InventosHumanos)$, mesmo que entre as duas classes existam várias outras. Porém entre as classes *Carros* e *Peixes*, nenhuma relação de generalização existe, pois ambos são especializações de classes completamente diferentes. Neste caso, $D(Carros) \neq D(Peixes)$.

4.3 Transações de esquema

Como o número de relacionamentos existentes em um esquema de certa complexidade é relativamente grande, quase todas as alterações resultarão em algum tipo de violação dos invariantes. Nestes casos as operações não serão completadas, o que, no uso real, praticamente impediria o projetista de realizar qualquer alteração mais importante pois as operações não seriam executadas pelo sistema.

Para evitar este problema e permitir ao projetista uma possibilidade real de evolução de esquemas, é necessário o conceito de **transação de esquema**. Durante uma transação de esquema, o sistema mantém o mecanismo de invariantes fora de atuação, possibilitando ao projetista realizar uma série de operações de evolução de esquema e ao final de todas as operações desejadas encerrar a transação de esquema. Neste momento, o sistema verificará a correção do estado atual do esquema e consolidará ou não as operações realizadas durante a transação. Porém, mesmo

dentro de uma transação longa, a estabilização de uma versão só deve ser possível se ela atender a todos os invariantes definidos, para garantir um posterior retorno somente a contextos corretos.

Transações de esquema serão, tipicamente, de longa duração, variando de algumas horas até dias, muito comuns em ambientes de projeto. Para este tipo de transação, são necessários mecanismos especiais de controle de concorrência, que possibilitem um desenvolvimento cooperativo da modelagem do esquema. Barghouti e Kaiser [BAR 91] analisam vários destes mecanismos, comparando-os entre si.

Verificar os invariantes em um dado instante, para dar uma indicação clara da consistência atual do esquema durante uma transação, é uma operação importante para auxiliar o projetista. Ela faz-se necessária devido à complexidade que um esquema pode atingir, pois o projetista muitas vezes não terá o entendimento completo de como uma alteração se refletiu no esquema.

Outras opções podem ser dadas ao projetista durante uma transação, de acordo com o método de controle de concorrência escolhido, como, por exempo, desfazer uma alteração, definir "savepoints" e retornar ao último "savepoint".

4.4 Operações de Alteração

Nesta seção, cada uma das possíveis alterações realizáveis sobre um esquema é descrita. Estas operações dividem-se em quatro blocos distintos:

- Alterações na estrutura de uma classe: operações relativas à implementação de uma classe, ou seja, seus atributos.
- Alterações no comportamento de uma classe: operações relativas aos métodos associados a uma classe, que alteram o comportamento de seus objetos.

- Alterações em grafos de generalização: modificações que envolvem mais de uma classe pertencente à hierarquia de generalização definida no esquema.

- Alterações em grafos de agregação: modificações que envolvem os relacionamentos de composição entre classes diferentes.

As operações básicas que compõem cada bloco são descritas a seguir, com seus parâmetros e semântica de funcionamento. Para cada uma, é analisado como pode afetar os invariantes, permitindo o estabelecimento de regras que deverão ser atendidas para a alteração ser aceita como correta pelo sistema. Estas regras são importantes, pois são mais simples de serem verificadas que os invariantes, por se restrigirem a uma parte do esquema. Outras operações mais complexas podem ser definidas a partir destas, como sendo a combinação de duas ou mais operações. Por exemplo, a cópia de um atributo existente para outra classe é semelhante à operação de inserção de atributo nesta classe. Assim, para não estender excessivamente o trabalho, nem todas as operações possíveis e imagináveis foram descritas.

4.4.1 Alterações na estrutura de uma classe

4.4.1.1 Adicionar um atributo em uma classe

Esta operação adiciona um novo atributo A, com domínio D, em uma versão da classe C pertencente ao esquema. Como há modificação na descrição da classe, se a versão sobre a qual ela estiver sendo aplicada estiver estabilizada ou o usuário requerer, uma nova versão da classe é gerada. Para as instâncias, um valor "default" ou uma função de cálculo pode ser indicada pelo projetista para o novo atributo A. Caso este nada indique, as instâncias assumirão o valor $NULL$.

Para a operação ter sucesso, não pode existir outro atributo com o mesmo nome na versão de classe C. Se o atributo A redefinir um atributo A', com domínio D', anteriormente herdado, então $D \subseteq D'$. Caso a classe C possua uma subclasse

C'' e o atributo A estiver redefinido nesta subclasse com domínio D'', a relação $D'' \subseteq D$ deverá ser verdadeira. Caso o domínio D restrinja um atributo herdado, valores já armazenados nas instâncias poderão se tornar inválidos, por excederem a nova restrição.

O atributo A poderá alterar a resolução de um conflito de herança ou criar um para as subclasses. Em ambas as situações, poderá acontecer que, nas subclasses, o domínio do atributo seja alterado. Então, como as descrições foram alteradas, novas versões serão geradas e todos os valores armazenados nas suas instâncias e referências existentes em métodos poderão se tornar inválidos no contexto considerado.

Como exemplo, podemos ter o estado do esquema mostrado pela figura 4.1, onde a classe C_3 é subclasse de C_1 e C_2, que possui um atributo A com domínio nos números inteiros. C_3 herda este atributo e suas instâncias terão valores numéricos para este atributo. O projetista, então, inclui um atributo A na classe C_1 tendo o conjunto de caracteres como domínio, gerando uma nova versão desta, $C_{1'}$ e de suas subclasses, no caso $C_{3'}$. Um conflito de herança é criado na subclasse $C_{3'}$, pois passam a existir dois atributos com o mesmo nome herdados das superclasses. O projetista resolve este conflito escolhendo o atributo A de $C_{1'}$. Todas as ocorrências deste atributo nos métodos de $C_{3'}$ tornam-se inválidas, devendo ter seus valores adequados ao novo domínio, ou recebendo um valor "default" indicado ou através de algum método de coerção.

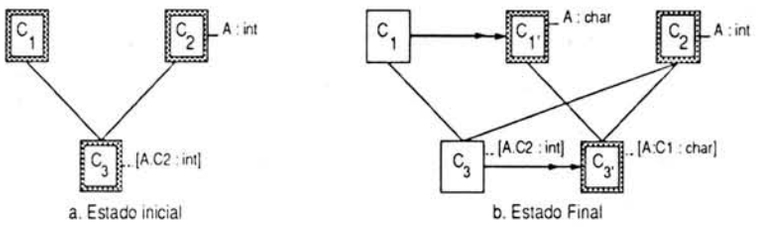

Figura 4.1: Exemplo de inclusão de um atributo

4.4.1.2 Remover um atributo de uma classe

Esta operação permite remover o atributo A da versão de classe C. Devido à conseqüente alteração na descrição da classe, novas versões da classe e de suas instâncias serão geradas pelo sistema, se estiverem estabilizadas, ou o usuário requerer.

A operação terá sucesso se o atributo A pertencer a versão de classe C. Se o atributo não participava de conflitos de herança nem redefinia algum outro herdado, ou seja, era o único no sub-grafo de generalização no qual estava inserido, novas versões da classe, das suas subclasses e instâncias serão geradas e os métodos que o referiam tornados inválidos.

Caso este atributo fosse o escolhido para resolver um conflito de herança, nova escolha deverá ser realizada. Se o domínio do atributo escolhido for diferente, a descrição das subclasses também será alterada e os métodos que faziam referência a este atributo tornar-se-ão inválidos no contexto. Redefinições do atributo nas subclasses podem ficar incorretas também, por deixarem de ser especializações do atributo herdado. Eventualmente, o conflito de herança pode deixar de ocorrer, também trocando o domínio do atributo nas subclasses. Nestes casos, todos métodos da classe que referiam o atributo tornam-se inválidos, bem como os métodos que enviavam mensagens aos primeiros, recursivamente.

Por exemplo, a partir da situação mostrada na figura 4.2a, onde o atributo A é redefinido na subclasse C_3 a partir da sua definição na classe C_2. Retirando o atributo A da classe C_2, o conflito de herança que existia na situação inicial desaparece e a subclasse C_3 passa a herdá-lo de C_1. A redefinição existente na nova versão da subclasse C_3 tornar-se-ia inválida, pois os números inteiros não formam um domínio especializado em relação aos caracteres. Neste caso a operação é rejeitada.

Já na situação inicial mostrada pela figura 4.2c, temos o mesmo atributo sendo redefinido corretamente para domínios especializados em cada subclasse. Assim, a remoção da redefinição do atributo A na subclasse C_2 mantém o esquema

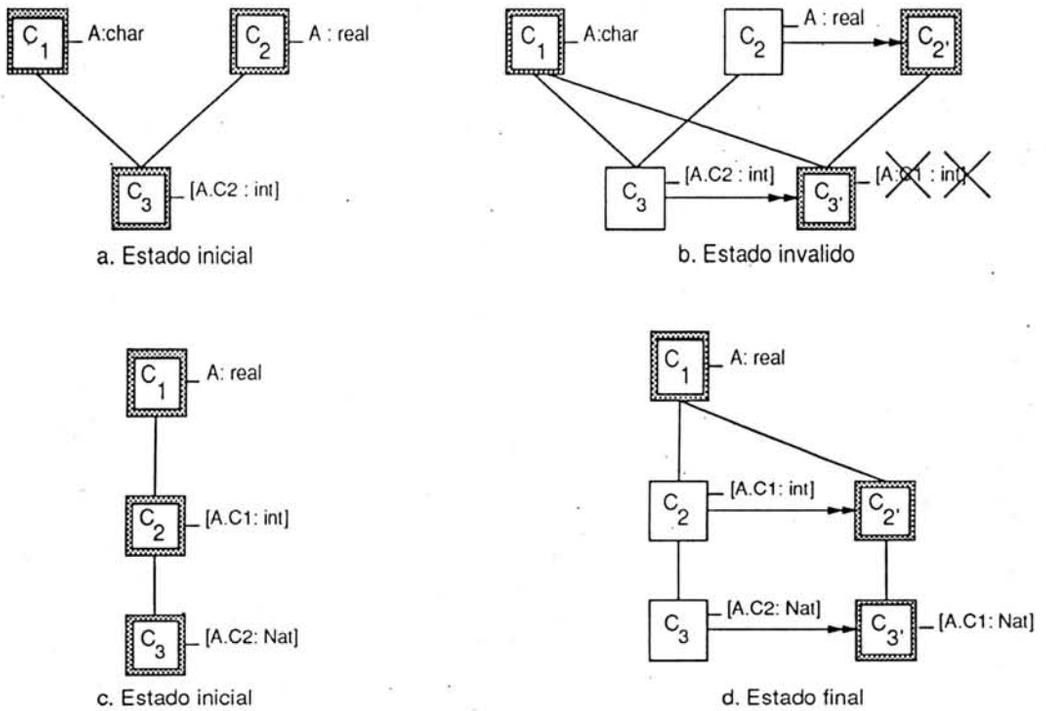

Figura 4.2: Exemplo da exclusão de uma atributo

correto e a operação tem sucesso. Uma nova versão da subclasse C_3 foi gerada porque ela teve parte de sua descrição alterada, pois o atributo A, embora continue redefinindo um atributo herdado, na nova versão $C_{3'}$ o faz de um atributo diferente da versão anterior.

4.4.1.3 Alterar o nome de um atributo

Esta operação permite a troca do nome de um atributo chamado A pertencente a versão de classe C para outro nome B. A descrição da classe é alterada e novas versões da classe e das instâncias são geradas, se já estiverem estabilizadas. A semântica da operação é a seqüência de remoção de um atributo A e inserção de um atributo B.

O atributo A deve pertencer à versão de classe C e não deve haver outro atributo com o novo nome, B, na descrição da mesma versão de classe C.

Para o antigo nome do atributo, A, as mesmas considerações da remoção de um atributo podem ser realizadas. Se ele resolvia um conflito de herança, outra

definição de A deverá ser escolhida para as subclasses. Se houver uma troca do domínio do atributo nestas subclasses, a alteração não será realizada. Se a sua remoção cria um conflito de herança para a classe C, esta deverá ser resolvida, procurando manter um domínio idêntico o esquema se manter correto.

Em qualquer caso, se a classe C passar a herdar uma definição qualquer de um atributo denominado A, as novas versões de instâncias assumirão um valor *NULL* ou um "default" que seja especificado pelo usuário, porque os valores antigos passarão a ser os valores do atributo B.

Para o novo nome do atributo, B, a operação é idêntica à inclusão de um atributo. Se ele alterar o domínio de um atributo herdado por uma de suas subclasses, os valores armazenados nas versões de instâncias de suas subclasses tornam-se inválidos, bem como redefinições do atributo e referências em métodos, no contexto da versão de classe alterada. Se o atributo B redefinir outro anteriormente herdado, seu domínio deverá especializar o domínio do atributo herdado.

Todas as referências existentes ao atributo nos diversos métodos podem ser alteradas para referências ao novo nome do atributo. Isto garante o funcionamento do método com a mesma semântica que possuia antes da alteração, exceto se a classe herdar um outro atributo A. Neste caso, o sistema não pode realizar esta substituição automática, podendo apenas alertar o projetista sobre eventuais problemas que possam ocorrer.

Por exemplo, digamos que o estado atual do esquema seja o mostrado pela figura 4.3a, onde a subclasse C_3 herda o atributo B de C_1 e resolveu o conflito de herança do atributo A optando por $A.C_2$. O projetista, então, altera o atributo A em C_2 para C, como pode ser visto na figura 4.3b. Assim, o conflito de herança que existia é desfeito e a classe C_3 passa a herdar o atributo A de C_1. Na classe C_2, todas as referências para o atributo A podem ser alteradas sem problemas para o novo nome, C. A classe C_3 passa a herdar o atributo C. Os antigos valores de A são colocados em C. Para o atributo A o projetista pode definir um valor "default"

diferente de NULL. As referências nos métodos de C_3 podem ou não ser trocadas para C, de acordo com o projetista.

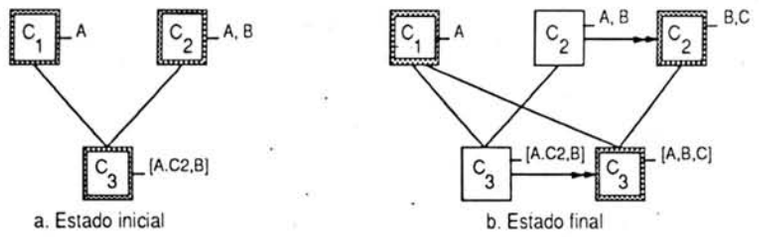

Figura 4.3: Exemplo de uma operação de troca de nome de um atributo

4.4.1.4 Alterar o domínio de um atributo

Esta operação permite alterar o domínio D de um atributo A pertencente a uma versão de classe C para outro domínio D'. Como a descrição da classe será alterada, se estiver estabilizada, novas versões da classe e das instâncias serão geradas.

O atributo A deverá pertencer à versão de classe C. Se o atributo for uma redefinição de um atributo herdado, o novo domínio deverá continuar sendo uma especialização do domínio herdado. Se subclasses redefinirem o atributo, esta deverá permanecer correta em relação à nova definição do domínio do atributo.

Três relações entre os domínios podem acontecer:

1 - $D' \subseteq D$: Neste caso, valores em instâncias poderão extrapolar o novo domínio, mais restrito que o anterior.

2 - $D \subseteq D'$: Com a relação inversa dos domínios, os valores das instâncias é que permanecerão todos corretos, mas os métodos poderão apresentar problemas ao operar valores pertencentes a um domínio mais aberto.

3 - $D \neq D'$: Então todos os valores e referências em métodos tornam-se inválidos no contexto onde a operação está sendo realizada. Isto ocorre por que estes domínios são completamente diferentes, um não é subtipo do outro mesmo se considerarmos todos os níveis existentes no conjunto de classes.

4.4.2 Alterações no comportamento de uma classe

Para as alterações de comportamento de uma classe, diversas modificações podem ser realizadas nos seus métodos. Estas são as que mais diretamente rompem com a transparência de alterações, pois são mudanças na interface com o mundo externo, que deverá readaptar-se ao sistema, seja um programa de aplicação ou um usuário final operando uma interface gráfica para o acionamento de métodos e escolha de objetos.

Sobre estas operações, várias verificações são realizadas procurando manter a correção do esquema na base de dados. Listas de métodos acionados e atributos da classe utilizados são mantidos para cada versão de método definida no esquema. Estas listas, obtidas através de uma análise sintática realizada sobre o código fonte do método, são utilizadas no teste dos invariantes comportamentais. Porém, mesmo com estas verificações, não é assegurado que o método funcionará corretamente, pois algumas modificações tem conseqüências semânticas que o sistema não consegue identificar em tempo de compilação devido aos mecanismos de ligação dinâmica e sobrecarga.

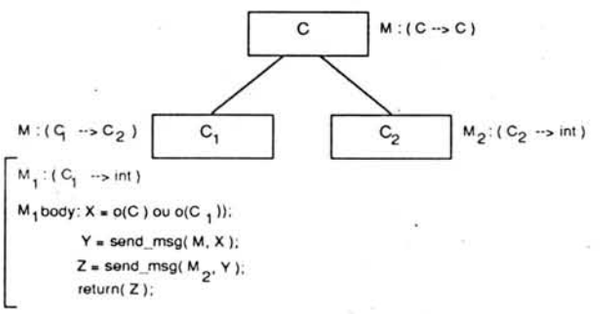

Figura 4.4: Seleção e verificação de métodos

Por exemplo, considerando a definição de classes e métodos mostrada na figura 4.4. No fonte do método M_1 podemos ver que a variável X recebe um objeto da classe C ou da classe C_1, de acordo com algum critério que não está claro, mas que não afeta a explicação de seu uso. Se para a variável X for atribuído um objeto da classe C_1, então o funcionamento do método estará correto, pois a seguinte seqüência de ativação ocorrerá:

$X = o(C_1)$; $Y = send\_msg(M, o(C_1))$; $Z = send\_msg(M_2, o(C_2))$; $return(o(int))$

Porém, se o atributo X receber um objeto da classe C, será acionado o método M como definido nesta classe, retornando para Y um objeto também da classe C. Em seguida, será enviada uma mensagem a esta classe para acionar o método M_2, que não está definido em C, o que resultará em um erro de execução.

$X = o(C)$; $Y = send\_msg(M, o(C))$; $Z = send\_msg(M_2, o(C))$; $ERRO$

Assim, a verificação de tipos, em tempo de compilação, não poderia indicar um erro, pois somente durante a execução será identificada a chamada indevida a um método inexistente na classe.

Se, em um momento posterior, o projetista remover a redefinição do método M da subclasse C_1, ele deverá receber uma notificação de que o método M_1 foi de alguma forma afetado, pois utilizava o método removido. Neste caso, será de responsabilidade do projetista alterar o método M_1, que agora sempre falhará, para uma alternativa correta.

O usuário é completamente responsável por manter corretas as aplicações que utilizam estes métodos, por não serem de nenhuma forma administradas pelo sistema.

Outro aspecto a considerar é quanto à validação de uma versão de método, que é garantida apenas para o contexto da versão de classe ou instância selecionada. O fato de uma versão de método ser inválida para determinada versão de classe não significa que o seja para todas as versões da classe. Por exemplo, para a situação mostrada na figura 4.5a, a versão M_1 é válida para todas as versões da classe C_1, pois considera os atributos A e B, existentes em todas elas. Na versão de classe C_{1m}, o atributo A é removido. No contexto desta classe, a versão de método M_1 torna-se inválida, o que é notificado ao projetista. Neste instante, se uma instância receber mensagem acionando o método, ela falhará por não existir uma versão válida do método associada à versão de classe. Uma solução possível é criar uma nova versão do método, $M_{1'}$, que utilize apenas o atributo B. Esta solução deverá ser realizada

pelo projetista. Nas demais versões da classe, se sofrerem alterações que gerem novas versões, por exemplo, incluir em C_1, o atributo X, a versão antiga do método ainda será válida, como mostrado em 4.5b.

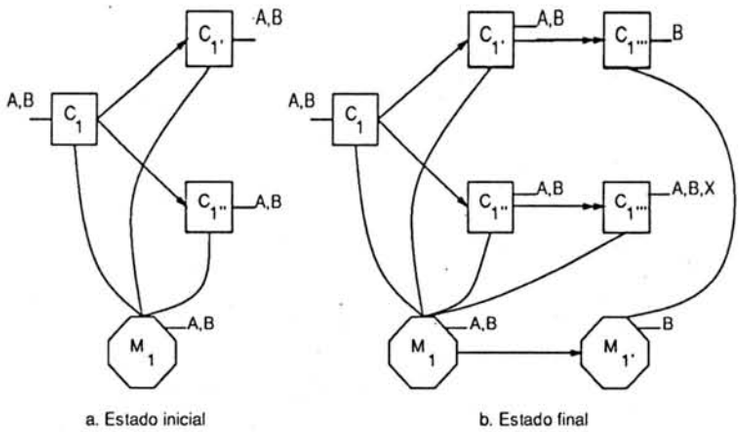

Figura 4.5: Exemplo de validação de versões de método

4.4.2.1 Adicionar um método em uma classe

Esta operação adiciona um novo método M em uma versão de classe C pertencente ao esquema.

Para a operação ser correta, não deve existir outro método com o mesmo nome na versão de classe C. Se o método M redefinir um outro método M' anteriormente herdado, a relação $M \prec M'$ deverá ser verdadeira. A redefinição que eventualmente exista em subclasses de C também deverá estar correta. O método M poderá alterar a resolução de um conflito de herança ou criar um para as subclasses. Neste caso, o projetista deverá resolvê-lo.

O novo método M poderá ter parâmetros e tipo de retorno com domínios mais especializados em relação a um método M' que passou a redefinir. Neste caso, as referências a ele em outros métodos, no contexto, poderão se tornar inválidas. Por exemplo, suponhamos que exista o método *altera_idade(idade: int): boolean* que permite a alteração do atributo *Idade* da classe *Pessoa* e retorna um valor booleano indicando se a operação foi realizada com sucesso ou não. Na subclasse *Adultos*, o atributo Idade é redefinido para o tipo *Maior_idade: set enum(18 .. 150)*

que restringe os valores aceitos para um intervalo de 18 a 150. Neste momento o projetista pode inserir um novo método na subclasse *Adulto* com a assinatura *altera_idade(idade: Maior_idade): boolean.* Caso um outro método envie uma mensagem para este, o tipo do parâmetro passado pode ser inválido, caso o valor seja menor que 18 ou maior que 150. Esta situação pode ser vista na figura 4.6.

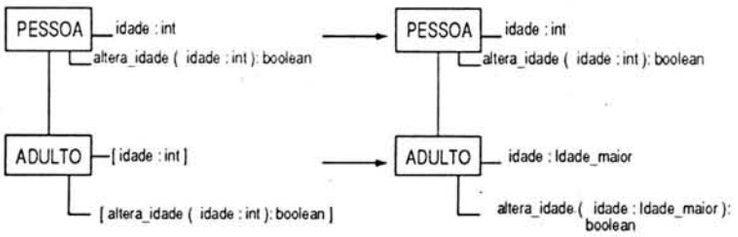

Figura 4.6: Exemplo de inclusão de um método

4.4.2.2 Remover um método de uma classe

Esta operação permite a remoção de um método M pertencente a uma versão de classe C.

O método deve pertencer à versão de classe especificada. Todos os métodos que o acionam tornam-se inválidos, se este não estiver redefinindo outro método M' herdado de uma superclasse. Caso o método M fosse a solução de um conflito de herança que persista ou a sua remoção crie um, o projetista deverá resolvê-lo.

4.4.2.3 Alterar o nome de um método

Esta operação permite que o nome N de um método M pertencente à classe C seja alterado para N'. Todas as chamadas deste método embutidas em outros métodos podem ser alteradas para o novo nome, se o projetista requerer. O sistema, automaticamente, deverá acrescentar o método em uma relação de nomes antigos e novos de métodos, para permitir que aplicações já escritas continuem funcionando. Outra possibilidade é o método M redefinir outro que agora passa a ser herdado pela classe, que poderá substitui-lo quando uma mensagem for enviada

para ele. O projetista recebe uma lista com as ocorrências de M em outros métodos para realizar a adequação necessária.

Problemas de conflito de herança, relacionados tanto com a novo nome quanto ao velho nome, podem ocorrer e devem ser resolvidos pelo projetista.

4.4.2.4 Alterar o código de um método

Para esta operação o projetista define o método M de determinada classe C que terá seu código fonte F alterado. O sistema não gera uma versão automaticamente, ficando sob responsabilidade do programador definir a necessidade de uma nova versão. Este mecanismo visa evitar a proliferação descontrolada de versões.

A lista de atributos e métodos referidos pelo método M é alterada para refletir a nova situação do código fonte, caso não haja geração de outra versão. Uma verificação léxica, sintática e semântica do novo código fonte deve ser realizada, porém não poderá ser completa devido ao mecanismo de ligação postergada, existente em linguagens orientadas a objetos. Esta verificação é disparada automaticamente pelo sistema.

4.4.2.5 Alterar o tipo de retorno de um método

O tipo de retorno D de um método M pode ser alterado para um novo domínio D'. Caso M redefina outro método M' herdado pela sua classe, a relação de redefinição deve ser mantida. Do mesmo modo, se em algumas subclasses o método estiver redefinido, a relação também deverá ser mantida.

Três relações podem ocorrer entre D e D':

1 - $D \subseteq D'$: as variáveis a que são atribuídos os valores de retorno do método possuem um domínio mais especializado e podem receber um objeto/valor que não pertença ao domínio original.

2 - $D' \subseteq D$: os métodos que o acionam continuarão corretos.

3 - $D \neq D'$: nesta caso todas as referências ao método M ficam inválidas.

Como definido na seção 3.5, o valor *VOID* é considerado como sendo diferente de todos os demais. Isto é coerente, pois se o funcionamento de um método for alterado de procedimento para função, ou ao contrário, todas as referências ao método que ocorram devem ser substituídas de modo adequado, pois onde não esperava-se valor de retorno, agora receberá um valor retornado, e onde se esperava algo, nada será retornardo.

4.4.2.6 Alterar domínio de um parâmetro

O domínio pd_i de um parâmetro P_i de um método M pertencente à classe C pode ser alterado para um novo domínio $pd'i$. Esta operação deve verificar se o novo domínio mantém a correção de métodos redefinidos nas subclasses e de uma eventual definição do mesmo método em uma superclasse.

Novamente três relações entre os domínios podem ocorrer:

1 - $pd_i \subseteq pd'_i$: ao enviar uma mensagem que acione o método M, o parâmetro poderá receber um objeto ou valor que, com a nova definição, excede o domínio especificado.

2 - $pd'_i \subseteq pd_i$: o método poderá ser executado com sucesso, pois os objetos ou valores recebidos pelo parâmetro ainda atenderão à nova especificação, menos restrita que a anterior.

3 - $pd_i \neq pd'_i$: todas as referências ao método M tornam-se inválidas.

4.4.2.7 Alterar o número de parâmetros

Esta operação permite incluir ou excluir um ou mais parâmetros na assinatura de um método. A alteração do número de parâmetros de uma assinatura torna incorreta qualquer redefinição existente para o método, inclusive o próprio

método, se ele redefinir outro herdado de uma superclasse, além de todas as chamadas para o método existentes na base de dados e em programas de aplicação.

4.4.3 Alterações em Grafos de Generalização

Diversas alterações em grafos de generalização podem ser realizadas pelos projetistas. Neste grupo de alterações, os conjuntos de superclasses e subclasses da classe considerada, respectivamente, $SP(C)$ e $SB(C)$ são utilizados como definidos na seção 3.4.

4.4.3.1 Acrescentar uma classe a lista de superclasses

Nesta operação é possível determinar que uma versão de classe S torna-se superclasse de outra versão de classe C, ou seja, a versão de C passará a ter mais uma superclasse. Para as regras de resolução de conflitos de nomes herdados, esta superclasse passa a ser considerada como a última da lista.

Como apenas a descrição da classe C é alterada, uma nova versão para ela e suas instâncias é gerada, se ela estiver estabilizada. A descrição da classe S não é alterada, portanto, não é necessário criar nova versão de S. Apenas seus objetos podem ser especializados em uma nova subclasse.

Obviamente, não pode haver uma ligação de generalização entre elas tal que $S \sqsubseteq C$, pois um ciclo não pode ser formado no grafo de generalização com esta operação. Se, inicialmente, a classe C estiver ligada diretamente a classe $GLOBAL$, S tornar-se-á a única superclasse de C.

Para cada atributo e método pertencente à classe S, as mesmas verificações da inserção de atributos e métodos em uma classe devem ser realizadas. Conflitos de herança podem ser criados ou sua resolução alterada, resultando em troca de domínio de atributos herdados em C e suas subclasses e na invalidação de valores armazenados em instâncias e referências realizadas em métodos, dentro do contexto que estiver sendo considerado.

4.4.3.2 Remover uma classe da lista de superclasses

Esta alteração permite retirar uma versão de classe S da lista de superclasses de uma versão de classe C. Esta operação resulta em mudanças na descrição da versão de C, suas instâncias e subclasses, para os quais novas versões são geradas. Versões em trabalho são realmente apagadas da base de dados. Versões estáveis são preservadas para manter o registro histórico do desenvolvimento. Sobre elas, apenas operações de consulta poderão ser realizadas.

A operação pode ser realizada se $S \in SP(C)$. Para cada atributo ou método definido na classe S que era herdado pela classe C, as considerações quanto a remoção de um atributo devem ser realizadas. Se um dos atributos ou métodos resolvia um conflito de herança, nova escolha deverá ser realizada. Os domínios dos atributos herdados podem ser alterados neste caso e tornar inválidos os valores nas instâncias e redefinições nas subclasses.

Se a classe S era a única superclasse de C, esta é ligada diretamente na classe $GLOBAL$ para garantir a conectividade do grafo de generalização.

4.4.3.3 Mover um atributo de uma superclasse para subclasses

Através desta operação é possível mover um atributo A definido em uma versão de superclasse S para um subconjunto $SB_A(S)$ de versões de suas subclasses que serão o destino do atributo. A descrição de todas as classes envolvidas é alterada e novas versões devem ser geradas pelo sistema, se já estiverem estabilizadas.

Para a alteração ocorrer, o atributo A deve estar definido na versão de S. O conjunto $SB_A(S)$ deve ser um subconjunto não vazio de $SB(S)$.

Na classe S o atributo é removido e todas as referências a ele em métodos definidos para a classe tornam-se inválidas, caso não redefinisse outro atributo A' herdado de suas superclasses. Nas classes pertencentes ao conjunto SB_A, se o atributo A já estiver redefinido, a operação não tem efeito, pois a redefinição na sub-

classe é mantida. Para o conjunto de subclasses que não recebem o atributo, ele é removido, tornando inválidas as referências a ele, exceto se:

1 - O atributo A já estava redefinido na classe S: neste caso, estas subclasses passam a herdar sua definição menos especializada, pertencente a uma superclasse de mais alto nível na hierarquia.

2 - O atributo A estava redefinido nas subclasses: então ele se mantém na descrição da classe, não mais redefinido, mas definido localmente.

Por exemplo, a partir do esquema definido na figura 4.7a, o projetista move o atributo A de C_1 para a subclasse C_2, conforme a indicação da flecha. A nova versão $C_{1'}$ não possui o atributo. A nova versão $C_{2'}$ possui o atributo, agora definido nesta classe. Uma nova versão $C_{3'}$ é gerada porque agora ela não mais herda o atributo A de C_1. A classe $C_{4'}$ mantém o atributo A, que antes redefinia.

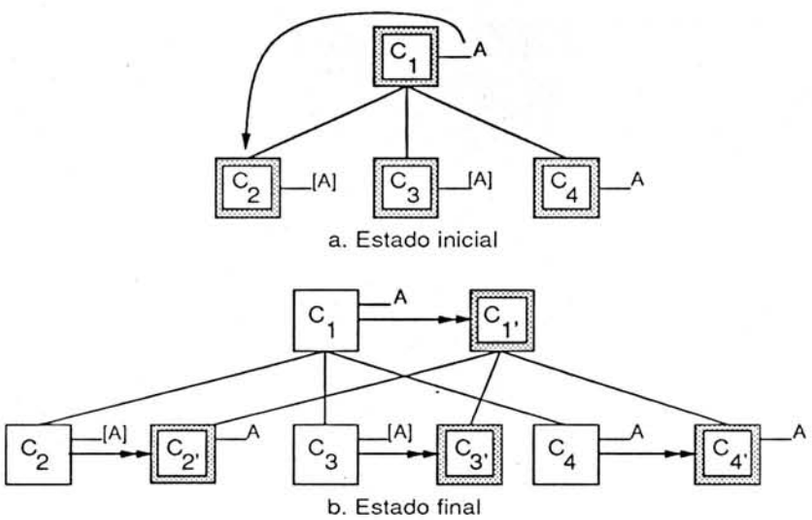

Figura 4.7: Exemplo de movimentação de um atributo para um nível inferior

4.4.3.4 Mover um atributo de uma subclasse para uma superclasse

Esta alteração possibilita que um atributo A pertencente a uma versão de classe C seja movido para uma de suas superclasses, S. Como ocorrem mudanças

na descrição das classes C, S e suas subclasses, novas versões são geradas, se as versões envolvidas estiverem estabilizadas.

O operação terá sucesso se o atributo A pertencer à definição da classe C e se a classe S pertencer ao conjunto das superclasses de C.

Na classe S o atributo está sendo inserido, portanto não deve haver outro com o mesmo nome. Caso outras subclasses de S, que não C, já possuíssem um atributo A definido, estes serão considerados como redefinições a nível de subclasses. Para as subclasses de S conflitos de herança podem ser criados e deverão ser resolvidos, bem como a resolução de um conflito anteriormente existente ser alterada. Em ambos os casos, o domínio do atributo herdado pelas subclasses talvez seja alterado e os valores armazenados tornem-se inválidos.

Na classe C o atributo é removido, mas passa a ser herdado. Assim, as referências existentes se mantêm corretas. Porém, problemas com conflito de herança podem ocorrer e o domínio do atributo A na própria classe C e nas suas subclasses ser alterado.

Por exemplo, na figura 4.8a, temos o estado inicial do esquema, com o atributo A definido nas classes C_2 e C_4. A alteração é realizada e o atributo A é movido de C_4 para C_1, conforme a indicação da flecha. Na classe C_2 ele é considerado como redefinido localmente. A classe C_3 passa a herdar um novo atributo. A classe C_4 perdeu a definição local do atributo A, mas passa a herdá-la.

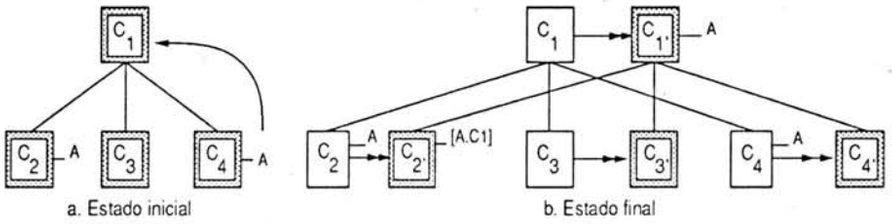

Figura 4.8: Exemplo de movimentação de um atributo para um nível superior

4.4.3.5 Mover um método de uma superclasse para as suas subclasses

Esta operação permite que o projetista mova um método M definido em uma classe S para um subconjunto $SB_M(S)$ de suas subclasses que receberão o método M. As conseqüências da movimentação são semelhantes às da movimentação de um atributo. O método M deve estar definido na classe S e o conjunto $SB_M(S)$ deve estar contido ou ser igual ao conjunto de subclasses de S.

Na classe S o método é removido e as referências a ele tornam-se inválidas. Nas subclasses pertencentes a $SB_M(S)$ se o método já estiver redefinido, o método M não será inserido, permanecendo a redefinição anteriormente existente. Para as subclasses que não recebem o método M, este ou é removido, ou passa a ser herdado de uma superclasse de S, ou permanece definido na subclasse.

Quanto ao método, pode ocorrer que ao referir-se a um atributo da classe redefinido localmente, o método atribua-lhe valores pertencentes ao domínio menos especializado, que excedem o domínio definido para o atributo na subclasse.

4.4.3.6 Mover um método de uma subclasse para uma superclasse

A movimentação de uma versão de método M definido na versão de subclasse C para uma versão de suas superclasses S é realizável através desta operação.

Para a operação ocorrer corretamente, o método deve estar definido para C e $C \in SB(S)$. O método não pode fazer referência a atributos e outros métodos apenas da subclasse C, pois estes não são reconhecidos na superclasse S.

Na classe S não pode existir nenhum outro método com o mesmo nome. Se houver redefinições do método nas subclasses de S, estas devem ser corretas. Problemas de conflito de herança podem ocorrer nas classes S, C e em outras subclasses de S.

4.4.3.7 Adicionar uma classe

Uma nova classe C pode ser adicionada ao esquema. O nome da classe deve ser informado e ser único no esquema. Seus atributos e métodos também devem ser informados, embora não sejam obrigatórios no primeiro momento, pois o usuário poderá acrescentá-los posteriormente.

O posicionamento da nova classe no grafo de generalização é informado através dos conjuntos $SB(C)$ e $SP(C)$. $SB(C)$ pode ser vazio. Se $SP(C)$ for vazio, será considerado como igual a $\{ \ GLOBAL \ \}$. Não pode ocorrer a formação de um ciclo no grafo de generalização.

Para cada atributo e método pertencente à classe, as mesmas verificações da inserção de um atributo ou método devem ser realizadas, visando a garantir a integridade do sistema.

4.4.3.8 Remover uma classe

Toda definição de uma classe pode ser removida do esquema se o projetista utilizar esta operação, bastando indicar a classe C desejada. Adicionalmente, o projetista pode definir se quer uma remoção em cascata de todas as subclasses de C. Caso não escolha esta opção, cada subclasse de C torna-se subclasse direta de cada superclasse de C, ou seja,

$$\forall\, C' \in SB(C), SP(C') = SP(C') \cup SP(C) \text{ e}$$

$$\forall\, C'' \in SP(C), SB(C'') = SB(C'') \cup SB(C).$$

Os atributos e métodos pertencentes a ela podem ser movidos para as suas subclasses, se o projetista desejar. Neste caso, nas subclasses onde os atributos ou métodos estiverem redefinidos, o sistema mantém esta redefinição. Caso o projetista opte por realmente removê-los, todas as referências em outros métodos tornam-se inválidas.

Atributos de outras classes que possuiam C como domínio são removidos de suas respectivas classes.

A classe C não é realmente removida da base de dados. Suas versões, instâncias e métodos permanecem para manter o registro do histórico de evolução do esquema. Apenas operações de consulta passam a ser possíveis sobre estas informações. Todas as outras não são mais aceitas pelo banco de dados.

4.4.4 Alterações em grafos de agregação

Como as relações de composição são mantidas através de atributos que possuem como domínio uma classe definida no esquema, as alterações na composição são realizadas através de modificações nestes atributos, já vistas na seção 4.4.1.

4.5 Extensões

Para auxiliar na manutenção da transparência de alterações, alguns mecanismos auxiliares podem ser incorporados. Estes mecanismos objetivam esconder da aplicação que alguma alteração foi realizada no esquema dos objetos, garantindo maior independência lógica de dados.

Quanto aos parâmetros, o ideal é que o envio de uma mensagem seja feito especificando os valores não apenas por posição, mas por nomeação. Assim, se algum for retirado, os parâmetros removidos poderão ser desconsiderados e a chamada continuará correta. Se houver acréscimo de parâmetros, o projetista pode definir um valor "default" que será utilizado caso a aplicação não envie valores para estes novos parâmetros. Uma espécie de assinatura máxima, contendo a união dos parâmetros existentes nas várias versões de cada método pode ser utilizada pela aplicação, permitindo que ela acione qualquer versão de método. Se houver mudança no domínio de um atributo, o projetista pode especificar uma função de coerção ou

o sistema aplicar uma automaticamente, para transformar valores de um domínio para outro.

Para métodos que forem renomeados, uma lista de nomes antigos e novos pode ser mantida para identificar a que entidade uma chamada antiga se refere. Para métodos removidos ou existentes apenas em versões posteriores da classe, uma lista de métodos alternativos pode ser indicada pelo projetista. O seletor do mecanismo de mensagens poderá fazer uso destas listas para modificar o método acionado para um alternativo, ou para o próprio, apenas com com um novo nome.

O domínio do tipo de retorno também pode ser modificado através de funções de coerção, que transformem o valor retornado para a aplicação em um valor correspondente ao domínio da variável ao qual este valor é atribuído.

Uma lista de atributos renomeados também pode ser mantida, com o mesmo objetivo da lista de métodos. Para atributos removidos ou não existentes em versões anteriores da classe, valores "default" podem ser mantidos para substitui-los caso sejam consultados inadvertidamente.

4.6 Conclusão

Ao longo deste capítulo o mecanismo de evolução de esquemas para o modelo de dados genérico foi apresentado. Este mecanismo baseia-se no conceito de invariantes, descritos no seu início. Em seguida, cada uma das operações possíveis foi definida, procurando-se deixar claro todas as suas possíveis implicações. Mecanismos auxiliares para garantir a transparência de alteração de esquema foram propostos, permitindo aumentar a independência de dados. No capítulo seguinte uma aplicação para este modelo genérico é apresentada e o necessário detalhamento e adaptação são realizados.

5 O AMBIENTE STAR

Neste capítulo é apresentado o ambiente STAR, sobre o qual um sistema para evolução de esquemas será definido e construído. O modelo de dados e os gerentes de versões e metodologia são explicados, tendo suas características mais relevantes para este trabalho detalhadas e comparadas com as desenvolvidas para o modelo genérico apresentado nos capítulos anteriores.

5.1 Características Principais

O ambiente STAR está sendo definido e implementado com o objetivo de atender às necessidades básicas para o desenvolvimento de ambientes para projeto de sistemas digitais assistidos por computador. Trata-se, portanto, de uma plataforma para desenvolvimento de ambientes específicos. Um sistema como o STAR deve permitir a integração de ferramentas de origens diversas, construídas para ele ou desenvolvidas para funcionar de forma independente de qualquer outra ferramenta, garantindo a integridade e consistência dos dados de projeto armazenados e a uniformidade de interfaces com o usuário. Assim, poderão ser desenvolvidos ambientes para diversas aplicações de projeto, arquiteturas e tecnologias utilizadas na sua realização [WAG 92].

O modelo de dados do ambiente STAR foi derivado a partir do modelo GARDEN [QUI 90]. Possui conceitos mais flexíveis e poderosos, visando facilitar a integração de ferramentas e o gerenciamento de diversas representações de um objeto de projeto onde os conceitos de classificação, generalização, herança e agregação são suportados de forma eficiente [WAG 91].

A evolução dos objetos de projeto é controlada através de um mecanismo para gerenciamento de versões incorporado ao modelo de dados básico [LAC 92], que permite armazenar diferentes formas de representação de um objeto:

- visões em diferentes níveis de abstração,
- alternativas de representação para um mesmo nível de abstração e
- revisões com correções em uma determinada alternativa.

As etapas de desenvolvimento do projeto podem ser especificadas em uma linguagem adequada que permite expressar as tarefas a serem realizadas através de um modelo baseado em pré e pós-condições, verificadas por um gerente de metodologias [WAG 91a, WAG 92].

5.2 Modelo de Dados

Um objeto no ambiente STAR é definido através de uma estrutura hierárquica, chamada de **esquema do objeto**, que possui diferentes níveis de abstração. Esta estrutura define o esquema de um objeto armazenado na base de dados. A figura 5.1 mostra a estrutura, definida em forma de uma árvore de generalizações, composta por nodos em vários níveis, que serão descritos a seguir.

5.2.1 Design, ViewGroups e Views

O esquema do objeto é basicamente definido por nodos pertencentes a uma destas três categorias. Cada esquema de objeto possui um único *Design*, zero ou mais *ViewGroups* e uma ou mais *Views* nas folhas da árvore, conforme a figura 5.1.

Design é um objeto de projeto único, como um microprocessador, ULA, registrador ou porta lógica. Ele representa o nodo raiz do esquema de um objeto definido no ambiente STAR. Cada *Design* é uma coleção de *ViewGroups* e *Views*, que herdam algumas de suas características.

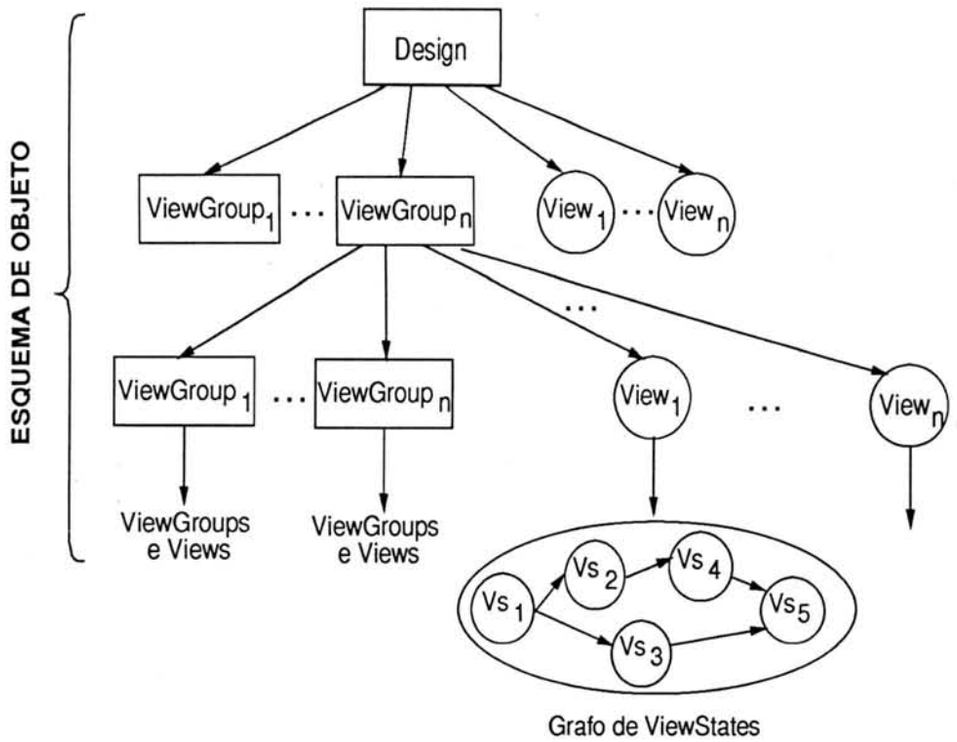

Figura 5.1: Modelo de Dados do STAR

ViewGroups são coleções de representações para um objeto de projeto que têm algumas propriedades em comum, definidas pelo usuário ou pela metodologia. Cada *ViewGroup* é uma coleção de outros *ViewGroups* e *Views*.

Views são as folhas da árvore no esquema de objeto e representam uma abstração deste em determinado nível.

Existem três tipos diferentes de *Views*:

- *HDLView*: são dedicadas a descrições comportamentais do objeto realizadas em um certo nível de abstração, normalmente linguagens de descrição de hardware ("Hardware Description Language - HDL").
- *MHDView*: são utilizadas para realizar descrições estruturais do objeto, em nível RT (Register-Transfer), lógico ou elétrico. Representam, deste modo, um objeto modular de forma hierárquica ("Modular Hierarchical Description").

- *LayoutView*: são orientadas a descrição geométrica das realizações físicas do objeto, armazenando o leiaute do circuito integrado ou de placas de circuito impresso.

5.2.2 ViewStates

ViewStates são os nodos onde os objetos de projeto realmente são armazenados, de acordo com a descrição herdada do esquema de objeto definido. *ViewStates* representam realizações reais dos objetos e estão organizados em um grafo de derivação, utilizado para manter a seqüência de alterações de um determinado objeto durante o seu projeto. Cada grafo de derivação está vinculado a uma *View*, de quem herda suas propriedades, como está mostrado na figura 5.1.

Existem três tipos de *ViewStates*, de acordo com o tipo de *View* a que está relacionada:

Os *HDLViewStates* e *LayoutViewStates* são descrições do objeto, cujas eventuais ligações com componentes internos são armazenadas na base de dados, mas não o modo como estas ligações são realizadas, isto é, o sistema tem conhecimento que um objeto possui conexões com outros objetos, mas não sabe quais saídas de um estão ligadas a quais entradas de outro. Em geral, indicam um ou mais arquivos onde a sua descrição detalhada está armazenada, inclusive as referências para outros objetos, tendo sido gerada por ferramentas automáticas de modelagem que reconhecem a estrutura interna deste arquivos.

Os *MHDViewStates* são puramente descrições estruturais do objeto, definidas explicitamente no sistema, inclusive quanto à semântica das interconexões entre seus componentes internos, modeladas através de *Ports* e *Nets*.

5.2.3 UserFields, Ports e Parameters

UserFields são atributos que podem ser especificados nos vários nodos de um esquema do objeto ou em *ViewStates*. O domínio de um *UserField* pode ser básico, pré-definido pelo sistema, como *integer, real, bit-vector, string, file* ou *time*; ou criado pelo projetista através de um dos construtores disponíveis: *record, array* ou *set*, bem como enumeração ou subconjunto de outro domínio especificado.

UserFields podem ser declarados como herdáveis pelos níveis inferiores ou não. A herança de *UserFields* pode ser "default", isto é, sua existência é herdada, mas seu domínio ou valor pode ser redefinido, ou estrita, quando nem o domínio nem o valor do *UserField* podem ser redefinidos nos níveis inferiores do esquema de objeto.

Ports representam os sinais de interface física entre objetos interconectados. Significam um conjunto de fios ou um único fio. Podem ser declarados em qualquer nível do esquema de objeto e em *ViewStates*. São herdados pelos níveis inferiores, sempre através de herança estrita, não podendo ser redefinidos nos níveis inferiores de nenhuma forma. Na sua definição, possuem três caracteristicas básicas: tipo de dados, direção e composição. Um tipo de dados pode ser associado a um *Port*. Este tipo pode ser *bit*, *integer* ou outro definido pelo usuário. O atributo *PortDirection* indica a direção da porta e pode ter os valores *in, out* ou *inout*, caso seja bidirecional. *Ports* dividem-se em *PortWires*, com apenas um fio para realizar uma conexão, e *PortBundles*, formados por um conjunto de *PortWires*, cuja existência depende diretamente da existência do *PortBundle* no qual estão definidos. Na composição de objetos de projeto, a ligação entre seus *Ports* pode ser definida através de *Nets*.

Parameters só podem ser definidos nos níveis do esquema de objeto. Possuem apenas nome e tipo. Quando o objeto for componente de outro no projeto realizado, valores serão atribuídos para estes *Parameters* de acordo com o tipo es-

pecificado. Caso sejam definidos como herdáveis são herdados pelos níveis inferiores da hierarquia de forma estrita.

5.2.4 Correlations

Correlations permitem a especificação de relacionamentos entre objetos, de acordo com critérios mantidos pelo usuário ou pela metodologia de projeto. Dois objetos quaisquer podem fazer parte de uma *Correlation*. A ligação entre eles possui uma direção e um modo. A direção indica uma dependência de existência. A notação $A \rightarrow B$ indica que B só pode existir se A existir. A direção pode ser bidirecional, direcionada ou não-direcionada. O modo especifica a ação a ser tomada em caso de uma remoção. O modo pode ser *protect* ou *remove*.

No modo *protect*, o objeto da esquerda não pode ser removido se o da direita ainda existir. Assim, se a ligação for bidirecional, nenhum dos dois poderá ser removido enquanto existir o relacionamento.

No modo *remove*, o objeto da direita é removido caso o da esquerda o seja. Se a ligação for bidirecional, a remoção de qualquer um dos dois, significa a imediata remoção do outro.

Em uma ligação não-direcionada, o modo não possui importância e nenhuma verificação é realizada pelo sistema.

A combinação destes dois atributos indica parte da semântica da *Correlation*, conforme pode ser visto, resumidamente, na tabela 5.1.

Tabela 5.1: Semântica de remoção de objetos de acordo com os atributos de *Correlations*

DIREÇÃO	MODO	REMOVER A	REMOVER B
$A \leftrightarrow B$	*Protect*	Não	Não
$A \leftrightarrow B$	*Remove*	Remove A e B	Remove B e A
$A \rightarrow B$	*Protect*	Não	Sim
$A \rightarrow B$	*Remove*	Remove A e B	Sim
$A - B$	*Protect*	Sim	Sim
$A - B$	*Remove*	Sim	Sim

Correlations também podem possuir *UserFields* e, para fins documentacionais, um critério de relacionamento. *Correlations* são versionáveis. Quaisquer alteração nas suas características básicas, significa a criação de uma nova versão da *Correlation.*

5.2.5 Repository, Library e Process

Repository é o conjunto de todos os objetos na base de dados. Ele é composto por uma lista de *Libraries* e outra de *Processes* vinculados.

Library é uma coleção de objetos de projeto. Cada objeto de projeto armazenado em uma base de dados STAR deve estar em uma *Library* específica.

Process contém informações de tecnologia e seu conteúdo é manipulado diretamente pelas ferramentas. Podem possuir referências a outros *Processes.* Se um *Process* não for definido junto ao *Repository* ou à *Library*, isso pode ser feito num *Design*, *ViewGroup* ou *View.* A partir do nodo em que é definido, ele é herdado pelos seus descendentes. A cada *Process*, uma lista de um ou mais arquivos de tecnologia está associada.

5.2.6 Outros Objetos

Além dos objetos descritos acima, alguns outros estão definidos para o modelo de dados do STAR, porém são utilizados durante a manipulação dos objetos

de projeto para estabelecer a configuração de objetos em ambientes com versionamento, não influindo na modelagem do esquema de objeto. Descrições destes objetos podem ser encontradas em [WAG 91, GRA 93, GRA 93a]. O esquema de objeto pode ser comparado ao esquema de classes em um modelo de dados orientado a objetos por possuir os mesmos conceitos de abstração.

5.2.7 Exemplo de um Esquema de Objeto

RISCO é um microprocessador de 32 bits que foi desenvolvido na UFRGS [JUN 90]. Executa uma instrução por ciclo, exceto referências para memória. Está dividido em quatro blocos principais: Operacional, de Controle, Interface de Validação e o Gerador de Clock. Suas principais características são:

- Dados, instruções e endereços de 32 bits.
- Um único barramento de 32 bits para endereços e dados.
- Possui 32 registradores de 32 bits, incluindo Program Counter, Stack Pointer e Processor Status Word.
- Instruções de desvio possuem sua execução atrasada em um ciclo.

Na figura 5.2 podemos ver parte do esquema de objeto para o microprocessador. Nela percebemos parcialmente a hierarquia de *Design*, *ViewGroups* e *Views* definida para o bloco operacional do RISCO. Alguns atributos estão colocados também para dar uma idéia mais completa da modelagem. Nela, temos os seguintes objetos:

- *Design* OP representa o bloco operacional. Existem outros *Designs* para o bloco de controle, ULA e registradores, por exemplo. Algumas alternativas estão colocadas no nível inferior, representadas por *Views* e *ViewGroups*.

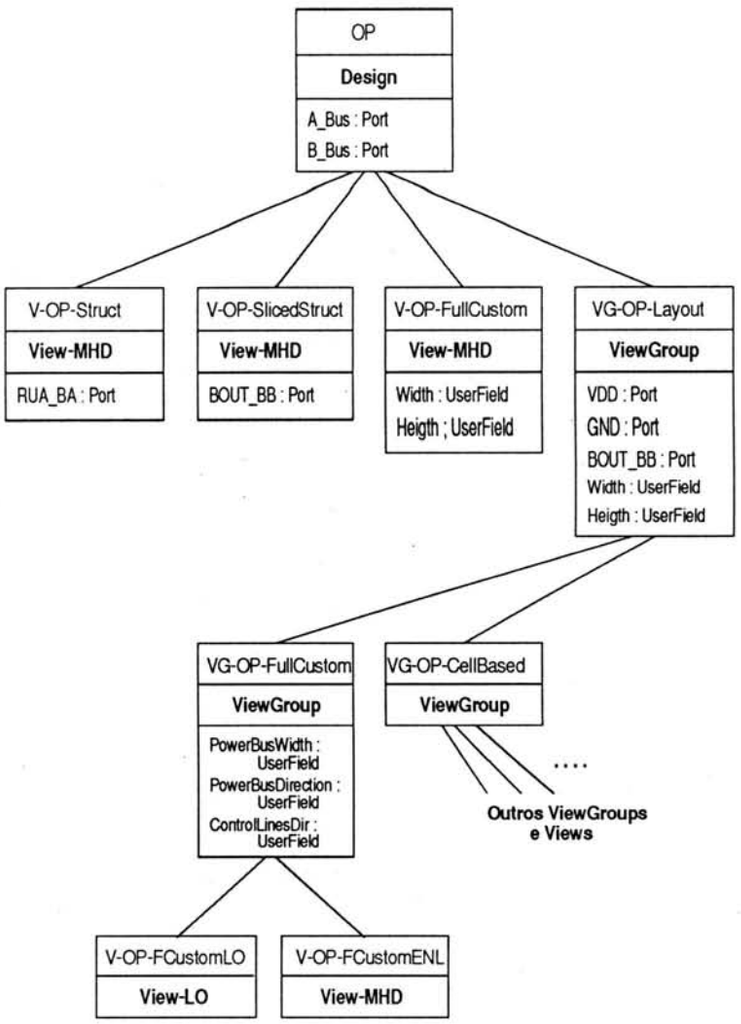

Figura 5.2: Definição do esquema de objeto para o bloco operacional do microprocessador RISCO no ambiente STAR

- *MHDView* V-OP-Struct representa os aspectos estruturais no nível RT, onde o bloco operacional será visto como uma interconexão de vários outros módulos, como ULA, registradores e barramento.

- *MHDView* V-OP-SlicedStruct é utilizada para manter a representação "sliced" do bloco operacional, isto é, a representação de apenas um bit, que será repetida 32 vezes.

- *MHDView* V-OP-FullCustom armazena a representação estrutural plana, na qual toda a hierarquização foi removida e há apenas componentes primitivos.

- *ViewGroup* VG-OP-Layout agrupa todas as representações do bloco operacional relacionadas ao seu leiaute.

- *ViewGroup* VG-OP-FullCustom agrupa todas as representações de leiaute realizadas utilizando uma técnica de projeto chamada "Full Custom".

- *ViewGroup* VG-OP-CellBased agrupa todas as representações de leiaute realizadas utilizando uma técnica chamada "Cell Based".

- *LayoutView* V-OP-FCustom-LO armazena a representação obtida para o leiaute no projeto "Full Custom".

- *LayoutView* V-OP-FCustom-ENL armazena uma representação do bloco operacional extraída, por engenharia reversa, a partir do leiaute obtido. Esta representação será comparada à existente no *MHDView* V-OP-FullCustom gerada no processo top-down de projeto, de modo a se verificar a correção do leiaute "full-custom", que foi feito manualmente.

Abaixo temos uma parte da declaração deste esquema de acordo com a linguagem de definição de dados do ambiente STAR [1].

```
OP:- Design
   has A_Bus:- Port
       has BitWidth       = { 31 .. 0 }
       has PortDirection  = inout
   has BOUT_BUS:- Port
       has Port Direction = in
   is generalization of
      {
       V-OP-Struct       ( View-MHD );
       V-OP-SlicedStruct ( View-MHD );
```

[1] Esta linguagem, chamada PLASMA, ainda não foi implementada. Há estudos em andamento para torná-la mais próxima de SQL [MEL 93], já pensando numa sintaxe mais geral que abranja, também, comandos de evolução de esquema.

```
        V-OP-FullCustom   ( View-MHD );
        VG-OP-Layout      ( ViewGroup )
      }
```

Esta definição cria um objeto *Design* com dois *Ports* e quatro descendentes no esquema do objeto. Para apenas um dos *Ports* foi definido um tamanho em bits. Estes *Ports* serão herdados por todos os objetos de nível inferior no esquema. O objeto não possui nenhum atributo, *UserField*, definido para ele.

```
OP:- VG-OP-Layout      :- ViewGroup
   has VDD             :- Port
       has Width       :- UserField: integer
       has Coordinates :- UserField: { integer, integer }
       has ImplemLayer :- UserField: Layer
   ....
   has Width, Heigth   :- UserField: integer
   is generalization of
      {
       VG-OP-FullCustom ( ViewGroup );
       VG-OP-CellBased  ( ViewGRoup )
      }

   Layer : string = { poly / diffusion / well / metal }
```

Esta definição cria um *ViewGroup* chamado VG-OP-Layout. Alguns *Ports* estão definidos, embora apenas um esteja aparecendo. Para este *Port*, três *UserFields* foram definidos, sendo um deles de tipo simples, outro um registro de dois números inteiros e o terceiro de um tipo definido pelo projetista. Além disso, o próprio *ViewGroup* possui dois *UserFields* do tipo inteiro, além de possuir dois subobjetos no esquema de objetos.

A definição completa deste exemplo pode ser encontrada em [WAG 92].

5.3 Gerente de Versões

5.3.1 Versões

Um mecanismo para gerência de versões está presente no ambiente STAR, visando a permitir a existência simultânea de várias representações de um mesmo objeto, criadas durante o desenvolvimento do projeto [LAC 92].

O modelo de dados do ambiente STAR permite representar as várias dimensões de evolução de objetos, **visões, alternativas** e **revisões**, através de dois níveis de versionamento:

- **Conceitual** : nos níveis do esquema de objeto, definindo alternativas de desenvolvimento e visões diferentes do mesmo objeto, através da modelagem em diferentes *ViewGroups* e *Views*.
- **Temporal**: no nível de *ViewStates*, através da criação de novos *ViewStates* é realizado o registro de revisões que o objeto sofre durante o desenvolvimento do projeto. O conjunto de *ViewStates* pertencentes a determinada *View* define um grafo de derivação, onde cada *ViewState* pode possuir um ou mais sucessores e um ou mais *ViewStates* predecessores. No nível de esquema de objeto, através da criação de novas versões de *Design*, *ViewGroups* e *Views*, forma-se uma sequência linear de versões em cada nodo do esquema de objeto.

Atributos podem ser definidos como **versionáveis** e **não versionáveis**. Esta característica é utilizada para definir quais atributos podem ou não serem alterados, dependendo do estado da versão, como será explicado na seção 5.3.3.

A figura 5.3 mostra um exemplo de versionamento de objetos no ambiente STAR. Nela podemos ver o esquema de um objeto e alguns de seus *ViewStates*. Neste instante, o esquema de objeto corrente é o definido pelo *Design* D_2, *ViewGroup* VG_2 e *ViewState* VA_4 e VB_2, na qual está ligado um grafo de derivação composto por

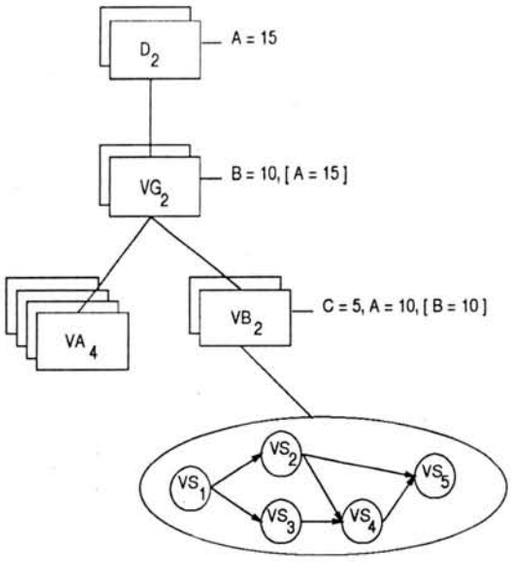

Figura 5.3: Exemplo de versionamento de um esquema de objeto no ambiente STAR

alguns *ViewStates*. *UserFields* herdados estão representados nos níveis de controle, representados entre colchetes. Isto ocorre com o *UserField A*, definido no *Design D* e com o *UserField B*, definido no *ViewGroup VG*. O processo de herança "default" é rompido pela atribuição de um novo valor ao *UserField A* na *View* VB_2.

5.3.2 Versão Corrente

O conceito de versão corrente é utilizado para definir o nodo sobre o qual as operações serão aplicadas, exceto se o usuário escolher explicitamente outra versão como sendo a corrente.

A operação de consulta a um objeto, retorna para o usuário sempre a versão corrente de cada nodo do seu esquema. Uma opção é consultar uma versão anterior, sem torná-la corrente. Quando uma nova versão é criada, ela torna-se a corrente. A mudança de versão corrente pode ser realizada apenas através de duas operações de seleção:

- **Total**: A partir da escolha de uma versão corrente em um determinado nível n da hierarquia, a seleção é propagada para os níveis superiores, selecionando sempre a versão associada à escolhida no nível n. Deste modo,

a descrição completa da versão, inclusive com os atributos herdados, é recuperada corretamente.

- **Parcial**: seleciona apenas a versão corrente em um determinado nível, sem propagar a seleção para os níveis superiores. Assim, apenas a descrição local da versão é recuperada, mantendo os atributos herdados com os seus valores atualizados.

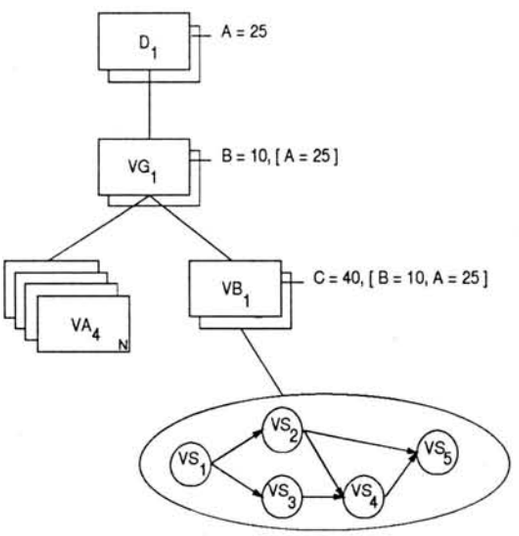

Figura 5.4: Exemplo de seleção total da $ViewVB_1$

Na figura 5.4, temos um exemplo de seleção total da primeira versão da *View VB*, realizada a partir da situação mostrada na figura anterior 5.3. Considerando que, quando esta versão foi criada, todos os níveis superiores da hierarquia continham apenas uma versão também, em todos estes níveis a versão corrente é alterada para a primeira. Os *UserFields* de cada nível são herdados com seus valores. Assim, na *View VB*, versão 1, onde o *UserField A* não é redefinido, ele possui o mesmo valor do nível de *Design*, onde está definido.

Na figura 5.5, temos um exemplo da mesma seleção, porém parcial. Neste caso, apenas no nível da *View VB* há mudança de versão corrente, permanecendo inalterados os demais níveis, dos quais os atributos são herdados. O *UserField A* é herdado com o valor definido da versão corrente do objeto *Design*.

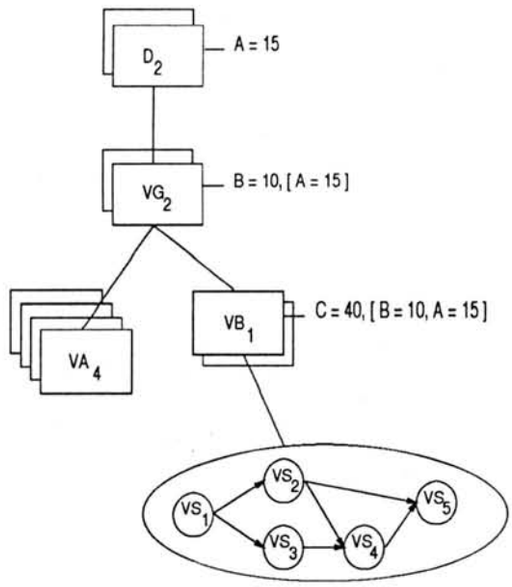

Figura 5.5: Exemplo de seleção parcial da $ViewVB_1$

5.3.3 Estado de Versões

De acordo com o grau de consistência, versões são consideradas como:

- **Em trabalho**: são versões sujeitas a alterações e remoção.
- **Estáveis**: possuem relativo grau de consistência, a sua consulta por outros projetistas já é permitida. Podem ser removidas, mas não alteradas. Versões de *Design*, *ViewGroups* e *Views* que já possuam versões sucessoras e *ViewStates* que possuam sucessores são consideradas como estáveis.
- **Consolidadas**: são versões que já atingiram alto grau de confiabilidade e que podem ser utilizadas como componentes de outros objetos pertencentes a qualquer projetista. Não podem ser alteradas nem removidas.

O projetista proprietário de cada objeto pode promover as versões manualmente.

Atributos, sejam *UserFields*, *Ports* ou *Parameters*, podem ser definidos como versionáveis ou não. Em nodos **em trabalho**, estes atributos podem ser modificados livremente sem gerar uma nova versão. Para nodos **estáveis ou consolidados**, atributos não versionáveis não podem ser alterados e a modificação nestes

resulta na criação de uma nova versão do nodo. Para *ViewStates*, a criação de uma versão significa criação de um novo *ViewState*, sucessor do que sofreu a alteração. Esta pode ser desde a modificação de valor até a inclusão ou remoção de atributos.

5.3.4 Notificação de Seleção

Sempre que o projetista realiza uma seleção total em um nodo, torna corrente um conjunto de versões mais antigas. Nesta situação, a versão corrente dos nodos filhos dos elementos deste conjunto são notificadas. Esta notificação indica que nestes nodos há a possibilidade de uma inconsistência, pois estavam herdando características, talvez diferentes, de uma versão que não está mais corrente. Enquanto uma versão notificada não for alterada, não surgirão problemas, pois ela está consistente com objetos de nível superior aos quais está ligada. Porém, caso ocorra alguma alteração que implique a criação de versões, do nodo notificado ou de algum nodo em nível inferior, a possível inconsistência deverá ser resolvida pelo projetista, que terá que escolher entre manter a versão atualmente válida, a corrente, ou trocar para uma versão ligada a corrente do nível superior, se existir uma. Manter a versão atualmente válida implica criar uma nova versão para aquele nodo, que passará a herdar uma nova versão corrente no nível superior. Eventualmente, pode não existir uma versão ligada ao nodo superior selecionado, por exemplo, se o nodo inferior tiver sido incluído em versões posteriores à selecionada.

Por exemplo, quando realizada a seleção total da *View* VB_1 mostrada na figura 5.4, foi notificada a versão de *View* VA_4, que se manteve corrente. Caso o projetista crie um *ViewState* associado a *View* VA, deverá escolher entre as versões de *View*, VA_1, ligada à VG_1, ou VA_4, a corrente. Após a escolha, a consistência do *ViewState* criado será verificado considerando a herança das caracteríticas herdadas como definidas nas versões escolhidas. Se o usuário escolher pela versão VA_1, mais antiga, uma nova versão, cópia da antiga, é criada como sucessora da atual. Esta solução é necessária por que as versões estão organizadas em um grafo linear de derivação.

Quando o projetista realizar uma seleção parcial em um nodo, como ele foi o único a ter a versão corrente alterada, apenas os seus objetos de nível inferior serão notificados, pois se versões deles ou nodos inferiores forem criadas, a mesma escolha entre a versão corrente e uma outra, deverá ser realizada.

5.4 Gerente de Metodologia

Uma das principais facilidades que o ambiente STAR oferece para a execução de um projeto complexo é a possibilidade de definir um fluxo de atividades a serem executadas para obter ao final do trabalho um objeto consistente, correto e testado [WAG 91a], constituindo uma metodologia de projeto. A definição de uma metodologia de projeto é baseada em três principios: um fluxo de tarefas, o esquema de objeto e hierarquização de metodologias.

A definição de uma seqüência de tarefas é realizada pela especificação de pré e pós-condições envolvendo as tarefas realizadas e as propriedades dos objetos relacionados a elas. As regras que compõem o gerente de metodologia são compostas basicamente por três partes:

- tarefas que devem ser executadas quando o projeto atinge determinado estado;
- um critério para escolha entre atividades alternativas que podem ser aplicadas em um determinado objeto;
- representações do objeto que devem ser criadas quando da execução da tarefa corretamente;

Em todas estas partes, as propriedades do objeto, armazenadas através de atributos podem ser consideradas para estabelecer o estado desejado, o critério a aplicar ou a representação a ser criada.

Cada tarefa é especificada através de uma 5-upla *(nome, ferramenta, pré-condições, objetos gerados, pós-condições)*. A tarefa especificada pelo nome só é executada pela ferramenta indicada se as pré-condições forem atendidas. A tarefa só é completada e seus resultados armazenados se as pós-condições forem atingidas e os objetos gerados corretamente.

Por fazer referências ao esquema de objeto e atributos definidos para determinado objeto, as tarefas do gerente de metodologia receberão tratamento semelhante ao dispensado a métodos no modelo genérico apresentado nos capítulos anteriores.

Metodologias de projeto podem ser organizadas hierarquicamente. A partir de uma metodologia inicial, novas podem ser definidas, especializando o esquema de objeto, acrescentando objetos a este esquema, ou adicionando novas tarefas. Esta organização possibilita que:

- um projetista perceba apenas as representações de objeto e tarefas pertinentes a ele. Objetos e tarefas definidas em outras metodologias não são vistas por ele.
- O projeto pode começar tão logo uma metodologia inicial seja estabelecida. Posteriormente, novos objetos e tarefas poderão ser acrescentados sem prejudicar o trabalho já realizado.
- Novas ferramentas podem ser integradas ao ambiente sem perturbar as atividades de projeto já em andamento. Estas novas ferramentas poderão fazer uso de novos objetos acrescentados ao esquema de objeto, definido para a metodologia.

Um exemplo parcial de metodologias de projeto para o microprocessador RISCO pode ser encontrado em [WAG 92].

5.5 Conclusão

Neste capítulo as principais características do ambiente STAR foram descritas. Por suportar muitos dos conceitos de orientação a objetos descritos no modelo genérico, existe uma semelhança entre o esquema de objeto do STAR e uma hierarquia de classes em um modelo orientado a objetos. Do mesmo modo, as tarefas definidas em uma metodologia se assemelham aos métodos de uma classe, principalmente por também envolverem a utilização de atributos. No próximo capítulo, uma proposta de evolução de esquemas de objetos para o ambiente STAR será apresentada. Esta proposta é uma aplicação do modelo genérico desenvolvido nos capítulos anteriores. Outros modelos de dados com características semelhantes de generalização e agregação poderiam ser utilizados, a opção pelo STAR baseia-se no fato de ser, atualmente, um projeto que está em desenvolvimento através de um esforço conjunto entre o CPGCC-UFRGS e o Centro Científico da IBM.

6 EVOLUÇÃO DE ESQUEMAS NO AMBIENTE STAR

Neste capítulo o mecanismo de evolução de esquemas para o STAR é apresentado, como uma aplicação do modelo genérico desenvolvido anteriormente. Assim, tomando o esquema de objeto como um esquema de dados e as tarefas do gerente de metodologias como métodos, o mecanismo também se baseia em invariantes que são utilizados para validar a correção das modificações realizadas, cuja semântica está descrita abaixo.

Neste capítulo, o termo **objeto** refere-se a um dos seguintes tipos de objetos: *Design*, *ViewGroup*, *View* e *ViewState*. Por **atributo** entende-se *PortWire*, *PortBundle*, *UserField* e *Parameter*.

6.1 Invariantes no Ambiente STAR

Os invariantes para o ambiente STAR foram definidos de acordo com as propostas para o modelo genérico apresentado no capítulo 4. Estes invariantes procuram assegurar a integridade do banco de dados e a correção do esquema de objeto modelado e das tarefas de metodologias definidas pelos projetistas.

São os seguintes os invariantes do ambiente STAR:

1. Todos os objetos na hierarquia de generalização relacionam-se com apenas um objeto do nível superior, definindo uma hierarquia simples. Esta restrição provém da definição de um esquema de objeto com sendo estruturado em uma árvore de objetos.

2. Todos os objetos descendentes de um mesmo nodo, em um esquema de objeto, possuem nomes únicos. Como no STAR o nome de um objeto é obtido pela composição dos nomes de todos os seus objetos ascendentes, desde o nível de *Design*, basta garantir que para um determinado nível da árvore

todos os nodos irmãos possuam nomes únicos, que todos os objetos existentes na base de dados possuirão nomes únicos.

3. Todos os atributos definidos para um objeto possuem nomes únicos. Esta restrição permite a identificação única de um atributo de um objeto, seja ele *Port*, *Parameter* ou *UserField*.

4. Todos os atributos pertencentes a um objeto de nível superior, definidos como herdáveis, são atributos de seus objetos descendentes. Como o esquema do objeto é uma hierarquia em árvore, não há a possibilidade de surgir conflitos de nomes como ocorre quando há herança múltipla. Portanto, todos os atributos herdáveis, são realmente herdados pelos níveis inferiores da hierarquia.

5. Apenas *UserFields* herdados através do processo de herança "default" podem ser redefinidos para domínios mais especializados ou valores diferentes dentro do domínio especificado. Atributos herdados por herança estrita não podem ser redefinidos. Esta restrição provém da própria definição de herança "default" e herança estrita.

6. Todos os objetos referidos estão presentes na base de dados. Esta regra é a mesma integridade referencial existente para os bancos de dados relacionais, apenas adaptadas para o ambiente STAR.

7. Todo valor especificado para um atributo de objeto deve pertencer ao domínio associado à sua definição ou ser igual a NULL. Esta restrição de valores é básica em todos as linguagens de programação tipadas e sistemas de bancos de dados existentes e indica que o valor armazenado pertence ao domínio definido para ele, seja no objeto ao qual pertence, ou tenha sido definido em algum outro objeto ascendente, do qual é herdado.

6.1.1 Transação de Modelagem de Esquema

O conjunto de invariantes assegura a correção e integridade dos objetos armazenados na base de dados. Porém, como exposto na seção 4.3, muitas vezes será necessária uma série de operações para, a partir de um estado consistente, alcançar um novo estado consistente na base de dados. Assim, uma transação de modelagem pode ser aberta pelo usuário, suspendendo a verificação dos invariantes até o momento em que a transação seja finalizada, exceto se houver alguma operação de estabilização de uma versão. Neste caso, a hieraquia onde está inserida a versão será verificada de acordo com os invariantes, para garantir que apenas versões corretas sejam estabilizadas. Isto é especialmente importante porque assegura que posteriormente uma seleção desta versão resultará em um conjunto de objetos adequados e compatíveis entre si. Por se tratar de uma operação com duração considerável, esta transação deverá ser incorporada a um mecanismo de transações longas, adequado para o modelo.

6.2 Operações de Alteração de Esquemas de Objetos

Um conjunto de operações de alteração de esquemas de objetos está definido para permitir a sua modelagem e a de tarefas dentro do ambiente STAR. Estas operações formam a base de uma futura linguagem de alto nível para a definição de objetos no ambiente STAR.

No modelo de dados do ambiente STAR, o nome de um objeto é o identificador único reconhecido pelo usuário, embora internamente possua um "surrogate" para facilitar o seu armazenamento e reconhecimento. Assim, o nome dos objetos deve ser indicado quando da criação do objeto e não poderá ser alterado em nenhum instante pelo usuário. O nome completo do objeto é definido pela justaposição dos nomes de todos os objetos acima dele até o nível de *Library*, separados por ponto. Todas as referências posteriores ao objeto deverão ser realizadas utilizando seu nome

completo. Por exemplo, a partir da figura 5.2, o nome completo da *View* V-OP-FCustomLO é `OP.V-OP-Layout.VG-OP-FCustom.V-OP-FCustomLO`.

6.2.1 Operações no Esquema de Objeto

Na lista abaixo, as operações possíveis para cada tipo de objeto estão indicadas. Em itálico está o nome do objeto que está sendo definido, após, entre parênteses, seus dados básicos, e depois a lista de operações possíveis para o objeto. As chaves significam que o dado é obrigatório com opções, isto é, qualquer dos indicados é aceito, mas um deles deverá ser escolhido. Os colchetes significam não obrigatoriedade de definição. Neste caso, o sistema assumirá o valor *NULL*.

- *Library* (Nome)
  - Criação
  - Remoção
- *Design* (Nome, *Library*)
  - Criação
  - Remoção
- *ViewGroup* (Nome, { Nome_Design, Nome_ViewGroup }, [Critério])
  - Criação
  - Remoção
  - Troca de { Nome_Design, Nome_ViewGroup } de nível superior
  - Troca de Critério
- *View* (Nome, { Nome_Design, Nome_ViewGroup }, Tipo)
  - Criação
  - Remoção

  - Troca de { Nome_Design, Nome_ViewGroup } de nível superior

- *Parameter* (Nome, Domínio, Herdável, Versionável, Objeto)
  - Criação
  - Remoção
  - Troca de domínio
  - Troca entre herdável e não-herdável
  - Troca entre versionável e não-versionável.
  - Troca de objeto no qual está definido
- *UserField* (Nome, Domínio, Herdável, Versionável, Tipo de Herança, Objeto, [Valor])
  - Criação
  - Remoção
  - Troca de domínio
  - Troca entre herdável e não-herdável
  - Troca entre versionável e não-versionável.
  - Troca do tipo de herança
  - Troca de objeto no qual está definido
  - Troca de valor
- *Port* (Nome, Tipo, Objeto, Versionável, Direção, [Número de bits], [Domínio])
  - Criação
  - Remoção
  - Troca de objeto no qual está definido
  - Troca entre versionável e não-versionável.

- Troca de direção
- Troca do número de bits
- Troca de domínio

- *Correlation* (ObjetoEsquerda, ObjetoDireita, Direção, [Modo], [Critério])
  - Criação
  - Remoção
  - Troca de objetos
  - Troca de direção
  - Troca de modo
  - Troca de critério

6.2.1.1 Criação, Remoção e Movimentação de Objetos

Para todos objetos do ambiente as operações de criação e remoção estão definidas. Na criação, os dados básicos devem ser informados, incluindo o nome do objeto e o objeto de nível superior com o qual se relacionará. Neste nível, o nome fornecido deverá ser único. Valores "defaults" estão definidos para o caso do projetista não fornecer as demais informações.

Apenas versões em trabalho ou estáveis podem ser removidas. Uma versão estável removida não é retirada da base de dados, porém apenas operações de consulta poderão ser realizadas sobre ela, preservando, deste modo, o histórico do projeto. Uma versão em trabalho é realmente removida da base de dados e todas as referências realizadas a ela também são removidas. Em ambos os casos é realizada uma remoção em cascata, isto é, de todos objetos de nível inferior.

A movimentação de uma versão de objeto em nível superior só pode ser realizada para versões em trabalho. Como a movimentação de uma versão significa a remoção do local de origem e a inserção no destino, ela combina as semânticas destas

duas operações. O nome do objeto movido deverá ser único na sub-árvore onde estiver sendo colocado. Esta operação deve verificar se nenhum atributo herdado por herança estrita passará a ser redefinido e se a redefinição de atributos herdados por "default" está correta. A movimentação de um objeto que já possua subobjetos implica a destes também.

Estas operações devem ser utilizadas para criar o esquema de objeto de um *Design*. Por exemplo, na figura 6.1a, temos o *Design* D_1 que já possui um *ViewGroup* VG_1. O projetista aplica a operação de criação de um objeto para acrescentar ao esquema o *ViewGroup* VG_2, que possui um nome único em relação aos demais subobjetos de D_1. Sucessivamente, o projetista pode criar um esquema de objeto completo para o *Design*. Se desejar, pode remover objetos ou movimentá-los na hieraquia, colocando-os em outra posição do mesmo esquema ou, até mesmo, em outro esquema de objeto.

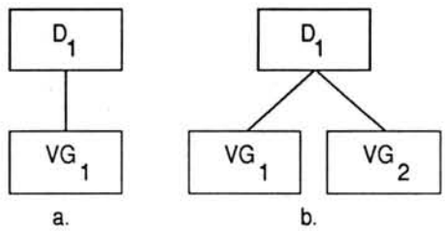

Figura 6.1: Inclusão de *ViewGroup* em um esquema de objetos.

6.2.1.2 Criação, Remoção e Movimentação de Atributos

Durante a criação de atributos, o sistema verifica se o atributo criado possui um nome único no objeto em que está sendo colocado. Caso seja declarado como herdável, redefinições que ocorram na sub-árvore serão verificadas para garantir que:

– caso o atributo esteja redefinindo um outro herdado, ele esteja sendo herdado "por default" e não por herança estrita e

– se a redefinição está sendo feita para um domínio igual ou subconjunto do herdado e caso haja redefinições deste atributo nos objetos inferiores, estas

também estejam corretas, de acordo com os invariantes. Valores que sejam fornecidos devem estar de acordo com os domínios estabelecidos.

Na remoção de atributos, as tarefas que referenciam o atributo são indicadas para o usuário, que poderá adequá-las à nova modelagem.

Para os atributos, a movimentação de um objeto para outro, significa a remoção do objeto de origem e inserção do atributo no objeto destino. Assim, ela só se realizará se os invariantes de redefinição estiverem atendidos, como explicado para a inserção. Este atributo poderá estar sendo referido em algumas tarefas. Neste caso, o usuário será notificado das tarefas afetadas.

Estas operações só podem ser realizadas em versões de objetos que estejam em trabalho. Se realizadas sobre versões estáveis ou consolidadas e o atributo for versionável, uma nova versão em trabalho será gerada. Elas não podem ser aplicadas sobre atributos não-versionáveis de versões estáveis ou consolidadas.

Na figura 6.2, temos um exemplo de inserção de um atributo em objeto. Inicialmente, temos três versões de objetos já estáveis (linhas em negrito) e o *UserField* A, com domínio igual a *Integer*, no *ViewGroup* VG_2. O projetista insere um *UserField* também chamado A, com domínio igual a *Real*, no *Design* D_1. Uma nova versão, em trabalho (linha simples) do *Design* e dos *ViewGroups* é gerada. O *ViewGroup* VG_1 passa a herdar A. O VG_2 mantem a definição local, cujo domínio é um subconjunto do domínio herdado. Para este segundo *ViewGroup* uma nova versão foi gerada, pois houve uma modificação na sua estrutura de herança. Como a versão anterior já estava estabilizada, a criação de uma nova versão foi necessária. A partir deste instante, outras modificações realizadas sobre $D_{1'}$ se refletirão imediatamente nas novas versões dos seus dois *ViewGroups*.

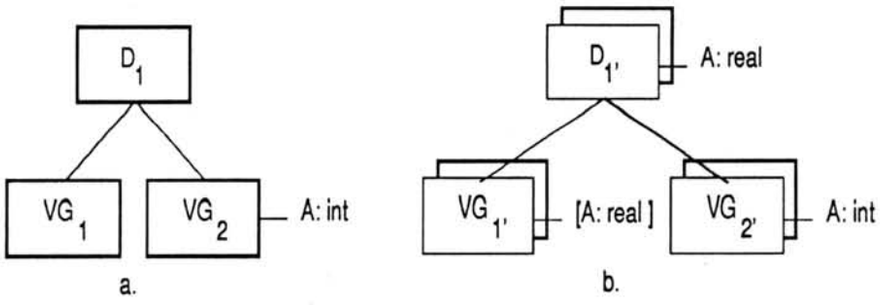

Figura 6.2: Inclusão de *UserField* em um esquema de objetos.

6.2.1.3 Alterações nas Características de Atributos

Herança

A passagem de um atributo de não-herdável para herdável significa a inclusão do atributo em todos os níveis inferiores da hierarquia, onde deverá ser verificada a correção de redefinições. Caso seja um *UserField*, um tipo de herança deve ser indicado, "default" ou estrita.

A troca de herdável para não-herdável significa a remoção do atributo em todos os objetos de nível inferior para os quais o atributo era herdado.

A troca do tipo de herança de um *UserField* pode realizar-se em dois sentidos: estrita para "default", onde nenhum problema ocorre, e de "default" para estrita. Neste caso, não pode haver redefinições nos objetos inferiores da hierarquia onde o *UserField* está definido.

Versionamento

A mudança de um atributo de não versionável para versionável pode ser realizada nos dois sentidos. Ela altera apenas a semântica de execução da operação de atualização de um valor de determinado objeto e não na estrutura do objeto, conforme explicado na seção 5.3.3

Digamos que em um objeto qualquer, como mostrado na figura 6.3, determinado atributo tenha sido declarado como versionável. Na versão i, esta carac-

terítica foi trocada, e ele passou a ser não-versionável. Na versão $i+2$, foi selecionada como corrente a versão inicial 1. Isto significa que uma nova versão foi gerada, $i+3$, com o atributo versionável. Embora um mesmo histórico de derivação contendo sucessivas versões que possuem o atributo versionável em algumas e não-versionável em outras possa tornar a modelagem confusa, para o sistema significa apenas que a partir de algumas a alteração do valor do atributo é permitida através da geração de uma nova versão e para outras esta alteração não é permitida, nem através da geração de uma nova versão.

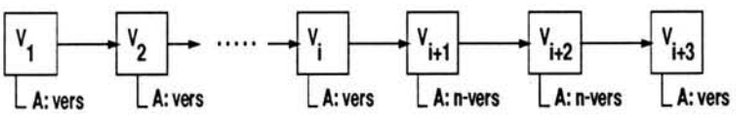

Figura 6.3: Problemas na alteração do versionamento de atributos

Domínio

Para a troca de domínio em *UserField*, um novo valor ou uma função de coerção pode ser indicada pelo usuário. As ocorrências do *UserField* em tarefas do gerente de metodologia serão notificadas pelo sistema para o usuário, que deverá realizar as correções necessárias.

O domínio de um *Port* é indicativo do tipo de dado que passará por ele, não havendo um valor diretamente associado. Sendo assim, a mudança de domínio não possui conseqüências maiores sobre instâncias armazenadas e pode ser realizada a qualquer instante.

Na troca de domínios em *Parameter*, o usuário pode indicar um valor "default" de acordo com o novo domínio ou uma função de coerção, que serão utilizados quando a referência ao objeto incluir um valor pertencente ao domínio anterior. Todas as ocorrências do objeto serão informadas ao usuário, que poderá, manualmente, alterar os valores passados como parâmetro.

Direção e Tamanho de *Ports*

A alteração do número de bits, que só é válida para *PortBundles*, implica a necessidade de alterar o número de *PortWires* componentes do respectivo *PortBundle*. Caso haja diminuição, o usuário deve indicar uma lista dos *PortWires* que serão removidos na nova versão. Se houver um aumento, uma lista de novos *PortWires* poderá, opcionalmente, ser fornecida pelo usuário.

Na alteração de direção de um *Port*, as direções de todos os seus *PortWires* são automaticamente alteradas, mantendo a coerência da estrutura definida.

6.2.1.4 Cópia de Objetos e Atributos

Uma operação de cópia de objetos e atributos já existentes, baseada na criação de objetos também está disponível. Basta indicar o objeto ou atributo a ser copiado e o seu destino no esquema de objeto. Caso seja um objeto com níveis inferiores associados a ele, estes níveis poderão ser copiados também. Todas as versões sucessoras à indicada são copiadas automaticamente. O objeto copiado deverá atender às especificações da inserção no destino.

6.2.1.5 Alterações em *Correlations*

Como as alterações permitidas em *Correlation* não afetam de nenhum modo os invariantes definidos para o ambiente STAR, elas podem ser realizadas com sucesso a qualquer momento, de acordo com a conveniência do projetista. Estas mudanças irão alterar apenas a semântica de remoção de objetos, que após a modificação de uma *Correlation*, poderá funcionar de um modo diferente ao que atuava anteriormente. Toda modificação realizada em um *Correlation* implica a geração de uma nova versão.

6.2.2 Operações no Gerente de Metodologias

As modificações nas tarefas definidas pelo gerente de metodologia não estão sendo descritas uma a uma por dependerem de uma definição mais aprofundada do mesmo. No entanto, algumas linhas básicas podem ser pensadas.

Quanto às mudanças no esquema de objeto, uma parte das relações já foi definida e deverá ser utilizada futuramente. São as relativas a remoção de objetos e/ou atributos, que eventualmente são referidos nas tarefas. Estas tarefas tornar-se-ão inválidas e deverão ser alteradas pelo projetista, para adequá-las ao novo esquema de objeto existente. Do mesmo modo em relação à modificação de domínios de atributos, que uma vez utilizados nas tarefas, poderão torná-las inválidas. Estas operações podem ser utilizadas também para criar um esquema de objeto a partir de outros, permitindo um reaproveitamento de definições já utilizadas e testadas em um projeto anterior.

Quanto às mudanças na metodologia de um projeto, cabe ressaltar que as tarefas que a compõem possuem um relacionamento bem mais fraco com o esquema de objeto que os métodos no modelo orientado a objetos, por não estarem ligadas diretamente a classes, não sendo de nenhum modo definidas sobre eles nem herdadas para níveis inferiores através da hierarquia de generalização. Assim, os efeitos de alterações realizadas sobre elas serão menores que os constatados na seção 4.4.1.3. Apenas as diferentes metodologias, mesmo alteradas, deverão manter uma hieraquia de especialização entre elas.

6.3 Implementação

A implementação do protótipo deste conjunto de primitivas está sendo realizada, assim como todo banco de dados do ambiente STAR, utilizando o sistema KRISYS [MAT 91] e a linguagem LISP [WIN 89]. Este protótipo está baseado em um gerente de tipos, responsável pela verificação de domínios, inclusive a avaliação

da relação de continência entre dois domínios, necessária para validar todas as redefinições de domínios existentes em objetos de nível inferior. Através deste gerente, uma biblioteca de tipos pode ser criada, permitindo a utilização de um mesmo tipo como domínio de atributos diferentes em vários objetos de um esquema.

Sobre este gerente, um conjunto de funções LISP permite a criação e manipulação dos objetos em uma base de dados. As operações são realizadas através da chamada de funções do gerente de versões, definido por [LAC 92], que está em fase final de implementação.

Posteriormente, prevê-se a definição de uma linguagem textual, ao estilo SQL, que permita a modelagem de esquemas de objetos de uma forma mais simples. Assim como a seleção de objetos poderá ser realizada de forma gráfica através de um "browser", que está sendo construído [MEL 93], para a criação e manipulação do esquema de objeto também deverá ser definida uma interface gráfica a ser integrada ao "browser", baseada em ambiente de janelas padrão OpenWindows.

6.4 Conclusão

Neste capítulo o mecanismo de evolução de esquema para o ambiente STAR foi definido. Este baseia-se no conceito de invariantes de esquema, utilizado por vários outros sistemas. Seu principal objetivo é permitir uma modelagem correta e flexível do esquema de objeto utilizado por determinada metodologia de projeto e dos objetos de projetos desenvolvidos pelas equipes de projetistas, bem como permitir a criação de novas metodologias de projeto a partir de outras já existentes. As operações permitidas, bem como sua semântica de operação foram definidas e estão sendo implementadas atualmente. Deste modo, os conceitos genéricos expostos nos capítulos anteriores puderam ser aplicados e, eventualmente, revisados. Uma implementação real está sendo realizada para testes futuros.

7 CONCLUSÃO

Este trabalho apresenta um mecanismo de evolução de esquemas para um modelo de dados orientado a objetos que não é específico de algum sistema, visando permitir sua adaptação posterior para um determinado ambiente. Este mecanismo, por utilizar versões de instâncias, classes e métodos, revela-se extremamente flexível e capaz de manter não só o histórico do desenvolvimento de determinada aplicação, como também de manter alternativas de um mesmo sistema que esteja sendo construído utilizando um banco de dados orientado a objetos.

Analisando os requisitos básicos para uma ferramenta de evolução de esquemas, como propostos na seção 1.1.1, podemos observar que:

- A integridade global do esquema é garantida a cada momento através de invariantes, assegurando que ao retornar a uma versão anterior, seu contexto estará correto e consistente, pois todas as verificações necessárias são realizadas pelo sistema. Estas verificações foram obtidas a partir dos invariantes especificados.

- A definição de contextos de esquema como forma de garantir que, em tempo de execução, não haverá uma interrupção por utilização de métodos e classe não compatíveis revela-se elegante. Os mecanismos auxiliares propostos complementam o sistema provendo soluções simples nos casos onde o acesso a objetos e a chamada a métodos não estiver atualizada, reduzindo a necessidade de modificações nos programas de aplicação, o que significará redução de custos no desenvolvimento e manutenção de tais programas. A utilização do conceito de transparência de alteração de esquemas para manter as aplicações que operam sobre o banco de dados funcionando sem a necessidade de modificações deve ainda ser mais explorada para reduzir a necessidade de alteração de pro-

gramas de aplicação que já estejam sendo utilizados e, portanto, já estão testados e são considerados confiáveis pela organização.

- A definição de regras para cada operação, obtidas a partir dos invariantes estabelecidos, é um recurso importante como forma de evitar uma sobrecarga do sistema, pois reduzem o espaço de verificação apenas para o conjunto de classes afetadas pela modificação, ao contrário da verificação dos invariantes que analisam todo esquema. A utilização de versões também auxilia neste sentido, pois permite o acesso a versões anteriores de objetos, enquanto novas estão sendo criadas, evitando a interrupção das aplicações.

- Procurou-se reduzir ao máximo a necessidade de intervenção humana, porém, em vários casos, apenas identifica-se o problema e deixa-se a solução para o projetista. Nossa idéia é explorar ao máximo as possibilidades de automação do processo. Para tanto, seria necessário investir em mecanismos que permitam reconhecer mais profundamente a semântica dos métodos definidos para uma classe.

- O máximo de operações possíveis são permitidas, inclusive algumas que rompem com a transparência de alteração de esquemas, procurando dar flexibilidade ao mecanismo. Ao conjunto de operações definidas, outras mais complexas podem ser acrescentadas combinando operações básicas já existentes. Por exemplo, ressurreição e cópia de objetos podem ser definidas a partir da criação de objetos.

- Como o mecanismo proposto não foi implementado, nenhuma interface com usuários está disponível. Para o STAR, prevê-se o desenvolvimento progressivo de interfaces mais amigáveis, pois inicialmente apenas um conjunto de funções estará disponível. Sobre este conjunto, deverá ser definida uma linguagem semelhante à SQL e, depois, uma interface gráfica.

A descrição do mecanismo genérico usando o modelo específico do STAR contribuiu para demonstrar sua aplicabilidade a outros modelos orientados a objetos e serviu como validação das idéias propostas, tendo permitido, inclusive, a correção de algumas falhas encontradas na definição inicial. Como será implementado, também permitirá demonstrar a sua viabilidade prática e o desenvolvimento de algoritmos adequados.

7.1 Contribuições Teóricas e Práticas

Este trabalho contribui nas pesquisas na área de evolução de esquemas em bancos de dados orientados a objetos por definir um mecanismo genérico extremamente flexível, baseado em invariantes e versões, que pode ser aplicado a modelos específicos. Na literatura disponível, não encontrou-se outro sistema que trata diretamente com versões de métodos de um modo simples. A definição de contextos de esquemas como forma de manter a transparência de alterações é um conceito que foi pouco explorado até em modelos que não permitem versões por identificar um subconjunto de classes, métodos e instâncias que podem ser afetados diretamente por uma modificação do esquema.

Quanto aos aspectos práticos, este trabalho contribui ao definir um mecanismo de evolução de esquemas para o STAR que está sendo implementado. Este mecanismo permitirá uma flexibilidade bem maior no desenvolvimento de projetos e a reusabilidade de descrições existentes em novos projetos.

7.2 Trabalhos Futuros

Uma série de investigações podem ser realizadas a partir deste trabalho. Talvez a mais importante seja estudar a possibilidade de incorporar a semântica dos métodos, que está descrita no código fonte dos métodos, para estender as verificações e modificações realizadas pelo sistema. Isto permitiria reduzir ainda mais a

necessidade de intervenção do projetista e incrementar a transparência de alteração de esquemas, pois até mesmo o código dos métodos poderia ser modificado.

A implementação deste mecanismo certamente não é uma atividade trivial. Algoritmos de verificação de invariantes, geração de versões e modificação de valores armazenados que evitem a sobrecarga do sistema mesmo havendo um grande número de classes, métodos e instâncias envolvidas são necessários para garantir a sua usabilidade, inclusive para aplicações onde o tempo de resposta é crítico.

A definição de uma interface gráfica que facilite a visualização do esquema e modificações claras do mesmo é importante para tornar seu uso mais simples e agradável.

A escolha de um mecanismo de transações longas que suporte a cooperação entre os projetistas enquanto mantém controle das operações que estão sendo realizadas é necessária para garantir a segurança quando o mecanismo for utilizado em um ambiente de desenvolvimento de sistemas onde atue uma equipe de projetistas.

As facilidades de gerenciamento de configurações do esquema, bastante úteis para simplificar o trabalho do projetista durante o desenvolvimento do esquema, podem ser ainda mais desenvolvidas. Assim, seria bastante útil definir um mecanismo de configurações que possua mais facilidades quanto à notificação de objetos através de vários métodos e, eventualmente, já realize a troca de componentes de modo automático, utilizando alguma especificação "default", pois facilitaria ainda mais o trabalho do projetista.

BIBLIOGRAFIA

[AME 90] AMERICA, P. A Behavioral Approach to Subtyping in Object-Oriented Programming Languages. **Philips Journal of Research**, Netherlands, v.44, n.2, July 1989.

[AST 76] ASTRAHAM, M. M. et al. System R: relational approach to database management. **ACM Transactions on Database Systems**, New York, v.1, n.2, p.97-137, June 1976.

[AST 79] ASTRAHAM, M. M. et al. System R: a relational database management system. **Computer**, Los Alamitos, v.12, n.5, p.42-49, May 1979.

[ATK 89] ATKINSON, M. et al. **The Object-oriented database system manifesto**. Le Chesnay: INRIA, 1989. 17p. (Rapport Technique *Altäir*, 30-89)

[BAN 86] BANERJEE, J.; KIM, H.H.;KIM, W.;KORTH, H.F. Schema evolution in object-oriented persistent databases. In: ADVANCED DATABASE SYMPOSIUM, Aug. 1986, Tokyo. **Proceedings**... Tokyo: Information Processing Society of Japan's Special Interest Group on Database Systems, 1986. p.23-31.

[BAN 87] BANERJEE, J. et al. Data model issues for object-oriented application. ACM Transactions on Office Information Systems, New York, v.5, n.1, p.3-26, Jan. 1987.

[BAN 87a] BANERJEE, J. et al. Semantics and implementation of schema evolution in object-oriented databases. In: ACM-SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, May 1987, San Francisco. **Proceedings**... New York: ACM, 1987. 509p. p.311-323.

[BAR 91] BARGHOUTI, N.S.; KAISER, G.E. Concurrency control in advanced database applications. **ACM Computing Surveys**, v.23, n.3, p.269-317, Sept. 1991.

[BEE 90] BEERI, C. Formal models for object oriented databases. **Data and Knowledge Engineering**, New York, v.5, n.4, p.353-382, 1990.

[BER 88] BERKEL, T. et al. Modelling CAD-objects by abstraction In: INTERNATIONAL CONFERENCE ON DATA AND KNOWLEDGE BASES, 3, June 28-30, 1988, Jerusalem. **Proceedings**...[S.l.:s.n.],1990.

[BER 83] BERNSTEIN; P.A.; GOODMAN, N. Concurrency control in distributed database systems. **ACM Transactions on Database Systems**, New York, v.13, n.2, p.185-222, Dec. 1983.

[BER 91] BERTINO, E. Object-oriented database management systems: concepts and issues. **Computer**, Los Alamitos, CA, v.24, n.4, p.33-47, Apr. 1991.

[BJÖ 89] BJÖRNERSTEDT, A. ; HULTEN, C. Version control in an object-oriented architecture. In: KIM, W.; LOCHOVSKY, F.H. (eds.). **Object-oriented concepts, databases and applications**. New York: ACM, 1989. p. 451-485.

[BRY 93] BRYANT, B. R.; CHANG, D. T.; LEE, T. **Integration of persistent** and relational database objects in CORAL In: Simpósio Brasileiro de Banco de Dados, 8., 12-14 maio 1993, Campina Grande, PB. **Anais**... Campina Grande: UFPB,CCT,DSC, 1993. p. 344-356.

[BUT 91] BUTTERWORTH, P.; OTIS, A.; STEIN, J. The GemStone object database system. **Communications of ACM**, New York, v.34, n.10, p.64-77, Oct. 1991.

[CEL 90] CELLARY, W.; JOMIER, G. Consistency of versions in object-oriented databases. In: INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASE , 16., 1990, Brisbane. **Proceedings**...[S.l.:s.n.],1990. p.432-441.

[CEL 91] CELLARY, W.; JOMIER, G.; KOSZLAJDA, T. **Formal model of an object-oriented database with versioned objects and schema.** 1991. 15p. (não publicado).

[CEL 91a] CELLARY, W.; VOSSEN, G.; JOMIER, G. **Multiversion object constellations for CAD databases.** Giessen: Justus-Liebeg-Universität-Giessen, 1991. 18p. (Bericht, 9105)

[CHE 90] CHEVAL, J.L. A Version model for object-oriented databases. Grenoble: Université Joseph Fourier, 1990. 16p. (Aristote Report, RAP007)

[CHO 88] CHOU, H.T.; KIM, W. Versions and change notification in an object-oriented database system In: ACM/IEEE DESIGN AUTOMATION CONFERENCE, 25, 1988. **Proceedings...**[New York:ACM],1988. 730p. p.275-281.

[COD 79] CODD, E.F. Extending the database relational model to capture more meaning. **ACM Transactions on Database Systems**, New York, v.4, n.4, Dec. 1979.

[COO 90] COOK, W. R.; HILL, W. L.; CANNING, P. S. Inheritance is not subtyping In: ANNUAL ACM SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES, 17, Jan. 17-19, 1990. **Proceedings...** New York: ACM, 1990.

[DAT 86] DATE, C.J. **Introdução a sistemas de bancos de dados.** 4.ed. Rio de Janeiro: Campus, 1986. 513p.

[DAV 86] DAVISON, J.W.; ZDONIK, S.B. A Visual interface for a database with version management. **ACM Transactions on Office Information Systems**, New York, v.4, n.3, p.226-256, July 1986.

[DEU 91] DEUX, O et al. The O_2 system. **Communications of ACM**, New York, v.34, n.10, p.35-48, Oct. 1991.

[DIT 87] DITTRICH, K.; GOTTHARD, W; LOCKEMANN, P.C. DAMOKLES. The Database system for the UNIBASE software engineering environment. **IEEE Data Engineering**, New York, v.10, n.1, p.37-47, 1987.

[FAU 91] FAUVET, M.-C. **Modelling and managing versions and histories in an object oriented environment.** Grenoble: Université Joseph Fourier, 1991. 14p. (Aristote Report, RAP015).

[FOR 92] FORNARI, M. R. **Um Estudo sobre evolução de esquemas em bancos de dados.** Porto Alegre: CPGCC da UFRGS, 1992. 98p. (Trabalho Individual, 250)

[GRA 93] GRAZZIOTIN, H. **Definição de configurações para o ambiente STAR.** Porto Alegre: CPGCC da UFRGS, 1992. (Dissertação de Mestrado)

[GRA 93a] GRAZZIOTIN, H.; GOLENDZINER, L. G.; WAGNER, F. R. **Gerente de configurações para o ambiente STAR.** In: Simpósio Brasileiro de Banco de Dados, 8., 12-14 maio 1993, Campina Grande, PB. **Anais...** Campina Grande: UFPB, CCT, DSC, 1993. p. 98-112.

[GRO 92] GROβ-HARDT, M. VOSSEN, G. **CLOOD: A Class-less model for object-oriented design databases.** Giessen: Justus-Liebeg-Universität Giessen, 1992. 17p. (Bericht, 9208).

[HAM 81] HAMMER, M.; McLEOD, D. Database description with SDM: a semantic data model. **ACM Transactions on Database systems**, New York, v.6, n.3, p.351-386, Mar. 1981.

[JUN 90] JUNQUEIRA, A.; SUZIM, A.; MARCON, C.; DOSSA, M.; CLETO, L. **RISCO - Multiprocessador CMOS 32 bits.** Porto Alegre: CPGCC da UFRGS, 1990. 24p. (Relatório de Pesquisa).

[KAT 90] KATZ, R.H. Toward a unified framework for version modeling in engineering databases. **ACM Computing Surveys**, New York, v.22, n.4, p.375-408, Dec. 1990.

[KIM 89] KIM, W.; BERTINO, E.; GARZA, J.F. Composite Objects Revisted. In: ACM-SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, May 31-June 2, 1989, Portland, Oregon. **Proceedings...** New York: ACM, 1989. p.337-347.

[KIM 90] KIM, W. **Introduction to object-oriented database systems.** New York: MIT-Press, 1990.

[LAC 92] LACOMBE, J.C.M. **Modelo de Versões para o Ambiente Star.** Porto Alegre: CPGCC da UFRGS, 1992. 134p. (Dissertação de Mestrado)

[LAI 79] LAINE, H.; MAANAVILJA, O.; PELTOLA, E. Grammatical Database Model. **Information Systems,** Uketer-UK, v.4, p.257-267, 1979.

[LAL 90] LALONDE, W.R.; PUGH, J. R. **Inside Smalltalk** 1.ed. New Jersey: Prentice-Hall, 1990.

[LAM 91] LAMB, C.; LANDIS, J.;ORENSTEIN, J.; WEINREB, D. The ObjectStore database system **Communications of ACM**, New York, v.34, n.10, p.51-63, Oct. 1991.

[LEC 88] LECLUSE, C. et al. Object-oriented database systems In: ACM-SIGMOD, 7, March 1988, Austin. **Proceedings...** New York: ACM, 1988. p.424-433.

[LI 88] LI, QING; McLEOD, D. Object flavor evolution in an object-oriented database system. In: CONFERENCE ON OFFICE INFORMATION SYSTEMS, Mar. 23-25, 1988, Palo Alto. **Proceedings...** New York: ACM, 1988. p. 265-275.

[MAI 86] MAIER, D.; STEIN, J.; OTIS, A.; PURDY, A. Development of an object-oriented DBMS. **SIGPLAN Notices,** New York, v.21, n.11,p.472-482, Nov.1986. Trabalho apresentado na Object Oriented Proggramming, Systems and LAnguages, 1986, Portland, Oregon.

[MAT 91] MATTOS, N. M. **An approach to knowledge base management.** Berlim: Springer-Verlag, 1991. 247p. (Lectures Notes in Computer Science, 513)

[MAT 93] MATTOS, N. M.; MEYER-WEGENER, K.; MITSCHANG, B. Grand tour of concepts for object-orientation from a database point of view. **Data and Knowledge Engineering,** Netherlands, v.9, n.3, p.321-352, Jan. 1993.

[MCK 90] McKENZIE, E.; SNODGRASS, R. Schema evolution and relational algebra **Information Systems,** Uketer-UK, v.15, n.2, p.207-232, 1990.

[MCL 88] McLEOD, D. A learning-based approach to meta-data evolution in an object-oriented database. In: INTERNATIONAL WORKSHOP ON OBJECT-ORIENTED DATABASES SYSTEMS, 2, Sept. 1988, Bad Münster, RFA. **Proceedings**. Berlim: Springer-Verlag, 1988. p.219-224. (Lecture Notes in Computer Science, 334)

[MEL 92] MELTON, J. (ed.) **Database language SQL**. ISO, 1992. 593 p.

[MEL 93] MELLO, Ronaldo **Uma Interface gráfica para o ambiente STAR**. Porto Alegre: CPGCC da UFRGS, 1993. (Dissertação de Mestrado em andamento)

[MOT 90] MOTZ, R.; FONSECA, D. From model based specifications to object-oriented prototypes. In: CONGRESSO DA SOCIEDADE BRASILEIRA DE COMPUTAÇÃO, 1990, Vitória. **Anais**... Rio de Janeiro: SBC, 1990. 611p. p.460-475.

[O_2T 91] O_2 TECHNOLOGY **O_2 Application Designer's Manual**. Paris:O_2 Technology July 1991. 327p.

[PEN 87] PENNEY, D.J.; STEIN, J. Class modification in the GemStone object-oriented DBMS. **SIGPLAN Notices**, New York, v.22, n.17, p.111-127, Dec. 1987. Trabalho apresentado na OOPSLA-87, Nov. 4-8, Orlando, FLA.

[QUI 90] QUINTANA, E. M. B. DE LA; ANNARUMMA, G. O.; NETO, P. M. **GARDEN:** the design data interface. Rio de Janeiro: IBM Rio Scientific Center, 1990. 28p. (Technical Report, CCR-007)

[ROD 92] RODDICK, J. F. SQL/SE: a query language extension for databases supporting schema evolution. **Sigmod Record**, New York, v. 21, n.3, p.10-16, 1992.

[RUG 91] RUGGIA, R.; MOTZ, R. A Schema manipulatiom mechanism for an OODB model. In: CONFERENCIA INTERNACIONAL DE LA SOCIEDAD CHILENA DE CIENCIA DE LA COMPUTACION, Oct. 1991, Santiago. **Proceedings**... Santiago:SCCC, 1991. p. 219-235.

[SEG 93] SEGAL, M.E.; FRIEDER, O. On-the-fly program modification: systems for dynamic updating. **IEEE Software**, Los Alamitos, CA, v.10, n.2, p. 53-65, March 1993.

[SKA 86] SKARRA, A.H.; ZDONIK, S.B. The Management of changing types in an object-oriented database. **SIGPLAN Notices**, New York, v.21, n.11, p.483-495, Nov. 1986. Trabalho apresentado na OOPSLA-86, Sept.29-Oct.2, 1986, Portland-OR.

[SMI 77] SMITH, J.M.; SMITH, D.P.C. Database abstractions: aggregation and generalization. **ACM Transactions on Database Systems**, New York, v.2, n.2, p.105-133, June 1977.

[VEN 91] VENTRONE, V.; HEILER, S. Semantic heterogeneity as a result of domain evolution. **SIGMOD Record**, New York, v.20, n.4, p.16-20, Dec. 1991.

[WAG 91] WAGNER, F.R. **The Data Model of the STAR Framework**. Porto Alegre: CPGCC da UFRGS, 1991. 27p. (Relatório de Pesquisa, 167).

[WAG 91a] WAGNER, F.R. Design Methodology Management in the STAR **Framework**. Porto Alegre: CPGCC da UFRGS, 1991. 18p. (Relatório de Pesquisa, 170).

[WAG 92] WAGNER, F. R. Modelling the **Design Methodology for the RISCO Microprocessor**. Porto Alegre: CPGCC da UFRGS, 1992. 38p. (Relatório de Pesquisa, 174).

[WEG 87] WEGNER, B. The Object-oriented classification paradigm. In: SHRIVER,B.;WEGNER,P.(eds.) **Research Directions in Object-Oriented Proggramming**. Cambridge:MIT Press, 1987. p.479-560.

[WIL 90] WILKINSON, K. LYNGBÆK; HASAN W. The Iris architecture and implementation. **IEEE Transactions on Data Knowledge and Data Engineering**, v.2, n.1, p.63-75, Mar. 1990.

[WIN 89] WINSTON, P. H.; HORN, B.K.P. **Lisp**. USA:Addison-Wesley, 1989. 611p.

[WOE 86] WOELK, D.; KIM, W.; LUTHER, W. An Object-oriented approach to multimedia databases. In: ACM SIGMOD CONFERENCE ON THE MANAGEMENT OF DATA , May 28-30, 1986, Washington, DC. **Proceedings...** New York:ACM, 1986.

[ZDO 86] ZDONIK, S. Maintaining consistency in a database with changing types. **SIGPLAN Notices**, New York, v.21, n.10, p. 120-127, 1986. Trabalho apresentado no Object-Oriented Programming Workshop, June 9-13, 1986, IBM Yorktown Heights.

[ZDO 90] ZDONIK, S. Object-oriented type evolution In: BANCILHON, F.; BUNEMAN, P. (eds.) **Advances in Database Programming Languages**. New York: ACM, 1990. p.277-88.

[ZIC 91] ZICARI, B. A Framework for schemas updates in an object-oriented database system In: DATA ENGINEERING, Apr. 8-12, 1991, Kobe, Japan. **Proceedings...** Tokyo: Information Processing Society of Japan's Special Interest Group on Database Systems, 1991. p.2-13.

Informática
UFRGS

Evolução de esquemas em bancos de dados orientado a objetos utilizando versões.

Dissertação apresentada aos Senhores:

Profa. Dra. Cláudia Bauzer Medeiros (UNICAMP)

Prof. Dr. Clesio Saraiva dos Santos

Prof. Dr. Jose Palazzo Moreira de Oliveira

Profa. Lia Goldstein Golendziner

Vista e permitida a impressão.
Porto Alegre, 29 / 09 / 93 .

Profa. Lia Goldstein Golendziner,
Orientador.

Prof. Dr. Ricardo A. da L. Reis,
Coordenador do Curso de Pós-Graduação
em Ciência da Computação.