UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS−GRADUAÇÃO EM COMPUTAÇÃO

Um Mecanismo de Notificação e Propagação de Mudanças para um Modelo de Versões

por
ANA CLARA GOTTFRIED DA FONSECA

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

Prof. Dr. Clesio Saraiva dos Santos
Orientador

Porto Alegre, julho de 2000

CIP − CATALOGAÇÃO NA PUBLICAÇÃO

Fonseca, Ana Clara Gottfried da

Um mecanismo de Notificação e Propagação para um Modelo de Versões / por Ana Clara Gottfried da Fonseca − Porto Alegre: PPGC da UFRGS, 2000.

93 F.: il.

Dissertação (Mestrado) − Universidade Federal do Rio Grande do Sul. Proprama de Pós−Graduação em Computação, Porto Alegre, BR−RS, 2000. Orientador: Santos, Clesio Saraiva dos.

1. Evolução de Objetos 2. Notificação de Mudanças. 3. Propagação de Mudanças. I. Santos, Clesio Saraiva dos. II. Título.

Agradecimentos

Agradeço

ao meu orientador Prof. Clesio Saraiva dos Santos, pela oportunidade concedida para a realização do mestrado e pela constante disponibilidade;

aos meus colegas Carina Dorneles, Adriana Roma e Daniel Notari pelas sugestões e correções do texto;

a todo o pessoal da biblioteca, especialmente à Ida;

às minhas amigas do PPGC Angela e Rose;

à colegas e amigas Andrea, Diana e Fernanda;

à minha querida tia Maria, aos meus primos e grandes amigos Ivo, Samarone e à minha prima Katia, os quais compartilharam suas casas comigo;

à minha mãe Helena pela permanente dedicação, ao meu pai Almir e aos meus irmãos Rogerio e Felicio pela força e incentivo;

a meu namorado Stefan pelo incentivo.

A todos vocês, quero dizer de coração, Muito Obrigada !

Gostaria de agradecer também ao CNPq pela concessão da bolsa.

Sumário

Listas de Abreviaturas.....6
Lista de Figuras.....7
Lista de Tabelas.....8
Resumo.....9
Abstract.....10
1Introdução.....11

2 Mecanismos de Notificação e Propagação de Mudanças.....14
2.1 Propostas encontradas em Projetos de Pesquisa.....14
2.1.1 Proposta segundo Atwood 16
2.1.2 Proposta segundo Chou e Kim 17
2.1.3 Proposta segundo Katz 17
2.1.4 Proposta segundo Mourad Chabane Oussalah.....18
2.1.5 Proposta segundo Luiz Fernando Soares.....19
2.2 Propostas encontradas em SGBDOOs 22
2.2.1 Itasca − Sistema de Gerenciamento de Banco de Dados para Objetos Distribuídos 22
2.2.2 ObjectStore − Sistema de Gerenciamento de Banco de Dados Orientado a objetos.....24
2.2.3 O2 − Sistema de Gerência de Banco de Dados Orientado a Objetos.....25
2.2.4 GemStone − Sistema de Gerência de Banco de Dados Orientado a Objetos.....26
2.3 Proposta segundo Gamma.....31
2.4 Conclusões 32

3 O modelo de versões de Golendziner 33
3.1 Objeto, Objeto Versionado e Versão 33
3.2 Identificadores de Objetos 34
3.3 Estado das Versões 34
3.4 Composição e Objetos Complexos 34
3.5 Configurações 35
3.6 Referências Estáticas e Dinâmicas 35
3.7 Herança por Extensão..... 35
3.8 Conclusões 36

4 Mecanismo proposto 37
4.1 Estratégias 39
4.1.1 Notificação Passiva e Notificação Ativa 39
4.1.2 Propagação.....40
4.2 Modelo de Classes 42
4.2.1 Classe User.....43
4.2.2 Classe NotificationSubscription.....43
4.2.3 Classe PropagationSubscription.....44
4.2.4 Classe Notification.....44
4.2.5 Classe NotificationPropagation.....45

4.3 Operações.....46
4.3.1 Subscrever Objetos para Notificação Passiva.....47
4.3.2 Cancelar Subscrição de Objetos de Notificação Passiva.....47
4.3.3 Consultar Mudanças.....48
4.3.4 Excluir Notificações.....48
4.3.5 Subscrever Objetos para Notificação Ativa.....49
4.3.6 Cancelar Subscrição de Objetos de Notificação Ativa.....50
4.3.7 Subscrever Objetos para Propagação.....50
4.3.8 Cancelar Subscrição de Objetos de Propagação.....51
4.3.9 Incluir Ocorrência de Notificações 51
4.3.10 Notificar Usuários 52
4.3.11 Propagar Mudanças.....53
4.4 Um Exemplo 53
4.4.1 Notificação Passiva 54
4.4.2 Notificação Ativa.....55
4.4.3 Propagação.....56
4.5 Conclusões 59

5 Implementação 65
5.1 Tabelas 65
5.1.1 Tabela Users.....65
5.1.2 Tabela NotificationSubscription.....66
5.1.3 Tabela PropagationSubscription.....67
5.1.4 Tabela Notification.....67
5.2 Operações do mecanismo.....68
5.2.1 subscribe_notificationp(oidSubject, oidObserver, Event) e unsubscribe_notificationa(oidSubject , oidObserver, Event).....69
5.2.2 notifyp(oidSubject, listoidObserver, user, event, timestamp).....69
5.2.3 consult_notification(oidSubject, oidObserver).....71
5.2.4 delete_notification(oidSubj, oidObserver).....71
5.3 Esquema Mechanism.....71
5.4 Exemplo de Aplicação 72
5.5 Conclusões 75

6 Conclusões 76

Anexo 1 Script para criação do Esquema Mechanism. 79
Anexo 2 Script para criação das tabelas do esquema Mechanism.....80
Anexo 3 Script para criação do pacote NotificationPassive_PAC 81
Anexo 4 Script das operações change_current, promote e derive 84
Anexo 5 Script para criação de triggers 86
Anexo 6 Script para criação de usuários e concessão de privilégios 88
Anexo 7 Script para criação das tabelas do Exemplo.....89
Bibliografia. 90

Listas de Abreviaturas

API	*Application Programming Interface*
BDOO	Banco de Dados Orientados a Objetos
BD	Banco de Dados
CAD	*Computer−Aided Design*
CASE	*Computer−Aided Software Engineering*
OID	*Object Identifier*
UML	*Unified Modeling Language*
SGBDOO	Sistema de Gerência de Banco de Dados Orientados a Objetos

Lista de Figuras

FIGURA 2.1 − Ambigüidade na criação automática de versões.....15
FIGURA 2.2 − Escopo da Notificação.....16
FIGURA 2.3 − Técnica *Version Percolation*.....16
FIGURA 2.4 − Criando uma Referência Inválida.....23
FIGURA 2.5 − Mecanismo de Notificação do Sistema O2.....26
FIGURA 2.6− *Observer* (notificação distribuída) extraída de [GAR 99].....31
FIGURA 3.1 − Objeto versionado e suas versões.....33
FIGURA 3.2 − Herança por extensão.....36
FIGURA 4.1 − Mecanismo de notificação e propagação.....38
FIGURA 4.2 − Ocorrência de Falha.....39
FIGURA 4.3 − Sentido da Propagação.....42
FIGURA 4.4 − Modelo de Classes.....42
FIGURA 4.5 − Classe *User* e suas Propriedades.....43
FIGURA 4.6 − Classe *NotificationSubscription* e suas Propriedades.....44
FIGURA 4.7 − Classe *PropagationSubscription* e suas Propriedades.....44
FIGURA 4.8 − Classe *Notification* e suas Propriedades.....45
FIGURA 4.9 − Classe *NotificationPropagation* − Propriedades e Operações 46
FIGURA 4.10 − Resultado da Operação *consult_notify*.....48
FIGURA 4.11 − Exemplo de Notificação Ativa.....53
FIGURA 4.12 − Projeto de Veículos.....54
FIGURA 4.13 − Criação de Versão com Notificação.....54
FIGURA 4.14 − Consulta − Notificação Passiva.....55
FIGURA 4.15 − Exclusão de Versão.....56
FIGURA 4.16 −Exemplo de Notificação Ativa.....56
FIGURA 4.17 − Propagação de Versões − Modo *restricted*.....57
FIGURA 4.18 − Propagação de Versões − Modo *extended*.....58
FIGURA 5.1 − Tabela *Users*.....66
FIGURA 5.2 − Tabela *NotificationSubscription*.....66
FIGURA 5.3 − Tabela *PropagationSubscription*.....67
FIGURA 5.4 − Tabela *Notification*.....68
FIGURA 5.5 − Procedimentos *subscribe_notificationp* e *unsubscribe_notificationp*.....69
FIGURA 5.6 − Procedimento *notifyp_all_observers*.....70
FIGURA 5.7 − Procedimento *notifyp*.....70
FIGURA 5.8 − Procedimento *consult_notification*.....71
FIGURA 5.9 − Procedimento *delete_notitification*.....71
FIGURA 5.10 − Esquema Conceitual Exemplo.....72

Lista de Tabelas

TABELA 2.1− Comparação entre Mecanismos Analisados.....21
TABELA 2.2 (a) − Comparação entre os Mecanismos Analisados 28
TABELA 2.2 (b) − Comparação entre os Mecanismos Analisados − Continuação 29
TABELA 2.2 (c) − Comparação entre os Mecanismos Analisados − Continuação 30
TABELA 4.1 − Eventos Controlados pela Notificação Passiva e pela Notificação Ativa 40
TABELA 4.3 − Eventos Controlados pela Propagação 41
TABELA 4.4 − Comparação entre os Mecanismos Analisados e o Mecanismo Proposto.....61
TABELA 4.5 (a) − Comparação entre os Mecanismos Analisados e o Mecanismo Proposto.....62
TABELA 4.5 (b) − Comparação entre os Mecanismos Analisados e o Mecanismo Proposto − Continuação 63
TABELA 4.5 (c) − Comparação entre os Mecanismos Analisados e o Mecanismo Proposto − Continuação 64
TABELA 5.1 − Tabela *Users*.....65
TABELA 5.2 − Tabela *NotificationSubscription*.....66
TABELA 5.3 − Tabela *PropagationSubscription*.....67
TABELA 5.4 − Tabela *Notification*.....68

Resumo

Um dos requisitos naturais na modelagem de diversas aplicações na área de banco de dados é a utilização de um mecanismo para controle de versões. Esse mecanismo fornece suporte a um processo evolutivo. Tal suporte permite armazenar os diferentes estágios de uma entidade em tempos distintos, ou sob diferentes pontos de vista. Estudos recentes nessa área mostram a importância de incorporar ao modelo conceitual de banco de dados, um mecanismo para auxiliar no controle da evolução de versões.

A evolução de versões apresenta problemas principalmente quando ocorre em uma hierarquia de composição. Por exemplo, se existem objetos compostos fazendo referência à objetos componentes que representam versões, então modificações nos componentes podem causar alterações nos objetos que os referenciam. Normalmente as ações relativas a essas modificações são a notificação ou a propagação de mudanças. Algumas propostas adicionam mecanismos de notificação e propagação ao modelo conceitual utilizado por aplicações não convencionais. Isso é importante porque mecanismos deste tipo auxiliam no controle da integridade de dados e na divulgação de informações sobre as mudanças realizadas no banco de dados.

O objetivo do trabalho aqui descrito é apresentar um mecanismo de notificação e propagação, que trata da evolução de dados, para um modelo de versões. É definido um modelo de classes com propriedades e operações que permitem manter e manipular subscrições de eventos referentes à evolução de objetos e versões e reagir diante da ocorrência destes eventos. Para atender os requisitos das diferentes aplicações, esta proposta especifica três estratégias. Cada uma delas apresenta diferentes funcionalidades: notificação ativa (enviar mensagens sobre mudanças ocorridas); notificação passiva (armazenar informações sobre mudanças ocorridas) e propagação (alterar o conteúdo do banco de dados automaticamente). Para validar o mecanismo proposto, uma implementação é apresentada para o sistema Oracle 8.

PALAVRAS−CHAVE: Evolução de Dados, Notificação de Mudanças, Propagação de Mudanças, Modelo de Versões.

TITLE: "A CHANGE NOTIFICATION AND PROPAGATION MECHANISM FOR A VERSION MODEL"

Abstract

One of the natural requirements in the modelling of several applications in the database area is the use of a version control mechanism. That mechanism offers support to an evolutionary process. Such support allows us to maintain different stages of an entity in different times, or under different point of view. Recent studies in this area, show the importance of incorporating to the conceptual model of database, a mechanism to aid in the control of the evolution of versions.

The Version evolution presents problems mainly when it happens in a composition hierarchy. For instance, if there are composed objects making reference to component objects that represent versions, then modifications in the components can cause alterations in the objects that reference them. Usually the actions related to these modifications are the notification and the propagation of changes. Some proposals add notification and propagation mechanisms to the conceptual model used by non conventional applications. This is important because mechanisms of this type help in the control of data integrity and spreading information about changes in the database.

The purpose of this work is to present a notification and propagation mechanism, which consider the data evolution, for a version model. A model of classes is defined with properties and operations that allow us to maintain and to manipulate subscriptions of events referred to the objects and version evolution and to react on the occurrence of these events. To meet the requirements of different applications, this proposal specifies three strategies. Each strategy presents different functionalities: active notification (to send messages about changes occurred), passive notification (to store information about changes occurred) and propagation (to alter the content of the database automatically). To validate the proposed mechanism, an implementation it is presented for the Oracle 8 System.

KEYWORDS: Data Evolution, Change Notification, Change Propagation, Version Model.

1 Introdução

Os modelos de dados relacionais têm sido amplamente utilizados no desenvolvimento da tecnologia de banco de dados requerida para tratar aplicações de banco de dados de negócios [ELM 94]. Esses modelos, entretanto, apresentam certas limitações para tratar aplicações complexas, tais como:

- CAD (*Computer−Aided Design*);
- CAM (*Computer−Aided Manufacturing*);
- CIM (*Computer−Integrated Manufacturing*);
- CASE (*Computer−Aided Software Engineering*);
- banco de dados científicos;
- SIG (Sistemas de Informação Geográfico);
- banco de dados multimídia.

Essas novas aplicações possuem requisitos e características que diferem de outras aplicações de negócios tradicionais, incluindo estruturas mais complexas para objetos, transações de longa duração, novos tipos de dados para armazenar imagens ou itens textuais longos e definição de novas operações.

Outra característica desejável, em novos domínios de aplicações, é a manutenção de múltiplas versões de um mesmo objeto. Na área de CAD, por exemplo, é possível utilizar versões com vistas à manutenção das múltiplas alternativas de projetos desenvolvidos por vários usuários. Com a utilização de versões, surgem algumas necessidades como a de tratar a evolução destas versões.

O modelo de versões proposto em [GOL 95] é uma extensão do modelo de dados orientado a objetos. A proposta teve como objetivo permitir a definição e manipulação de objetos, versões e configurações, possibilitando a navegação através da hierarquia de herança e permitindo manter transparente a manipulação de versões, quando desejado. Sempre que os objetos são referenciados sem identificação explícita de uma versão, é considerada a versão corrente. Quando necessário, versões podem ser explicitamente identificadas através do seu OID.

Versões de objetos podem ser definidas como compostas de outros objetos simples ou compostos, os quais são chamados de objetos componentes. Quando objetos componentes evoluem (através de alteração em uma versão ou de derivação de uma nova versão), pode haver a necessidade de notificar os usuários do banco de dados ou até mesmo fazer alterações nos objetos compostos. Estudos recentes [URT 98, SOA 98] mostram que um mecanismo de notificação e propagação de mudanças deve ser incorporado ao modelo conceitual utilizado, para especificar aplicações não convencionais. Isto é importante para auxiliar no controle da integridade dos dados e informar os usuários sobre as mudanças realizadas por outros usuários.

O objetivo principal deste trabalho é estender o modelo de versões proposto em [GOL95] com um mecanismo de notificação e propagação de mudanças.

Um mecanismo de notificação e propagação de mudanças foi sugerido no modelo de versões [GOL 95] como trabalho futuro, para tratar somente a evolução de versões de objetos componentes. A proposta desenvolvida pelo presente trabalho, além de considerar este tipo de mudança, também se preocupa com as mudanças ocorridas em objetos e objetos versionados.

O termo evolução de dados é usado no texto para se referir às seguintes mudanças:

- inclusão de objetos, objetos versionados e versões;
- exclusão de objetos, objetos versionados e versões;
- modificação de objetos, objetos versionados e versões.

Segundo Okendo [ALL 97], um SGBDOO deve suportar algumas características, para serem utilizados por sistemas de *Workflow*. Uma dessas características é um mecanismo de notificação ativa que capacite a reação automática a mudanças com execução de ações necessárias para definição de esquemas e dados.

O suporte à interação entre usuários é um requisito importante para auxiliar em trabalhos cooperativos, podendo ser fornecido de duas formas, segundo [LIM 92, LIM 96, LIM 97, TOL 99]. Na primeira, a interação entre usuários é realizada através de um espaço de trabalho especial. Cada projetista tem seu espaço privado, mas pode trocar informações com outros projetistas através de um espaço de grupo onde itens de dados compartilhados podem ser armazenados e recuperados. Na segunda forma, a interação entre os usuários é realizada em tempo real. Uma ação de um usuário sobre um contexto compartilhado é imediatamente propagada para todos os membros do grupo.

O mecanismo desenvolvido neste trabalho visa a tratar a evolução nos dados não sendo avaliada a evolução de esquema. Esse mecanismo pode ser usado para auxiliar trabalhos cooperativos, uma vez presentes em banco de dados. Por exemplo, sistemas de *workflow* que interagem com banco de dados.

Visando atender às diferentes necessidades apresentadas pelos usuários, três estratégias são estabelecidas: notificação ativa (envio de mensagens), notificação passiva (armazenamento de informações sobre mudanças ocorridas) e propagação (alteração do conteúdo do banco de dados). O mecanismo adiciona novas estruturas ao modelo de versões: classes com novas operações e propriedades.

Esta dissertação está organizada da seguinte maneira: no capítulo 2, são expostas as principais características dos mecanismos propostos para solucionar os problemas surgidos com a evolução de versões. Esses mecanismos são apresentados por alguns projetos de pesquisa e por SGBDOOs comercias; uma comparação destes também é exibida. Ainda neste capítulo, é descrito o padrão *Observer* e um exemplo do seu uso em banco de dados. No capítulo 3, é apresentado o modelo de versões utilizado como base para o desenvolvimento deste trabalho. No capítulo 4, é descrito o mecanismo de notificação e propagação de mudanças para o modelo de versões apresentado no

capítulo 3, suas características, estratégias, estruturas e operações. Para ilustrar o uso do mecanismo proposto, alguns exemplos são descritos, bem como as etapas realizadas em cada um deles. No final deste capítulo, é feita a comparação do mecanismo com os demais analisados no capítulo 2. No capítulo 5, a implementação do mecanismo para o sistema de Banco de dados Oracle é descrita. Finalmente, no capítulo 6, encontram−se as conclusões e alguns aspectos desta proposta que poderão ser futuramente explorados.

2 Mecanismos de Notificação e Propagação de Mudanças

O termo notificação é usado em diversas áreas, tais como: sistemas *Workflow*, sistemas de *e−mail* [KHO 95], entre outras. Neste trabalho é usado o conceito de notificação de mudanças como um mecanismo de divulgação das informações referentes às mudanças no conteúdo do banco de dados, através de envio de mensagens ou *e−mails* para os usuários, ou o armazenamento persistente destas informações. O conceito de propagação de mudanças corresponde obrigatoriamente a ações que alteram o conteúdo do banco de dados. A notificação e a propagação ocorrem mediante a ocorrência de determinados eventos.

Neste capítulo, são descritos mecanismos de notificação e propagação de mudanças encontrados na literatura. A seção 2.1 apresenta uma descrição das propostas encontradas, os problemas apresentados por elas acerca da evolução de versões de objetos componentes, bem como as soluções apresentadas pelos projetos de pesquisa que incorporam tais propostas. A seção 2.2 destaca os mecanismos que tratam das mudanças e da ocorrência de determinados eventos disponíveis em SGBDs através de serviços, são descritos na seção 2.2. A seção 2.3 descreve o padrão de projeto comportamental, que trata mudanças ocorridas em componentes cooperativos.

2.1 Propostas encontradas em Projetos de Pesquisa

Quando existem objetos compostos, utilizando como componentes objetos que representam versões, modificações nos componentes, como a remoção ou surgimento de uma nova versão, podem causar alterações nos objetos que os utilizam. As ações a serem tomadas, relativas a modificações nos componentes são: notificação [CHO 86] e/ou propagação [KAT 90].

Notificação de mudanças é definida em Chou [CHO 86] como uma funcionalidade desejável em ambientes CAD, a qual reage a mudanças em versões de objetos divulgando informações sobre alterações no banco de dados. Propagação de mudanças segundo Katz [KAT 90], corresponde ao processo que cria novas versões automaticamente.

Com a utilização de mecanismos de notificação e propagação surgem alguns problemas que devem ser tratados. Dois deles são: ambigüidade e escopo de notificação e/ou propagação.

A ambigüidade pode ocorrer quando um objeto componente é referenciado por mais de uma versão do mesmo objeto composto. Para tratar disto, uma estratégia deve ser definida. Uma estratégia simples é definir somente um caminho de propagação, ou seja, somente uma versão composta é criada (no caso da propagação) ou notificada (no caso da notificação), mesmo tendo *n* versões compostas referindo ao mesmo componente. Outra estratégia, desta vez mais complexa, é possibilitar a propagação (ou notificação) a todas as versões compostas que referenciam o objeto componente. Neste caso, o sistema deve fornecer mecanismos para gerenciar os *n* caminhos que podem surgir na hierarquia de composição.

O problema da ambigüidade é ilustrado na figura 2.1. Uma nova versão do objeto b1 é criada e com isso novas versões de todos os objetos que referenciam b1. O problema acontece quando é desejável que somente a versão a3 seja criada.

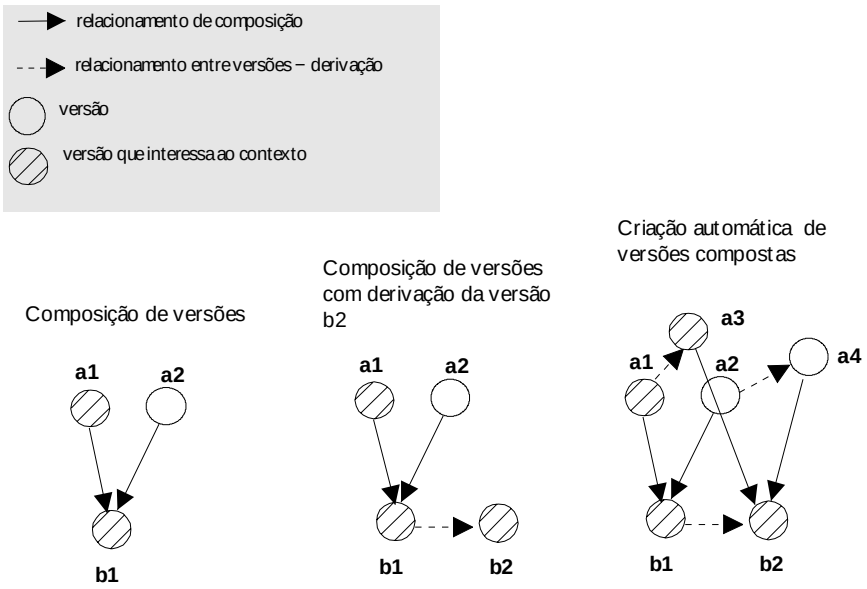

FIGURA 2.1 − Ambigüidade na criação automática de versões

O fato de uma versão fazer referência a outras versões, como ilustra a figura 2.2, em uma hierarquia de composição, de maneira recursiva, introduz o problema do escopo da notificação. Existem duas formas de definir o escopo da notificação:

1. somente as versões que fazem referência direta (no exemplo b1) à versão modificada são notificadas;

2. todas as versões que fazem referência direta ou indireta à versão modificada (no exemplo b1 e a1) são notificadas.

O argumento a favor da primeira opção baseia−se na idéia de que a versão que faz referência direta pode ou não ser alterada para acompanhar a alteração de seu componente. Somente se o usuário da versão optar pela alteração, o processo de notificação continua para o segundo nível de referência e, assim, sucessivamente.

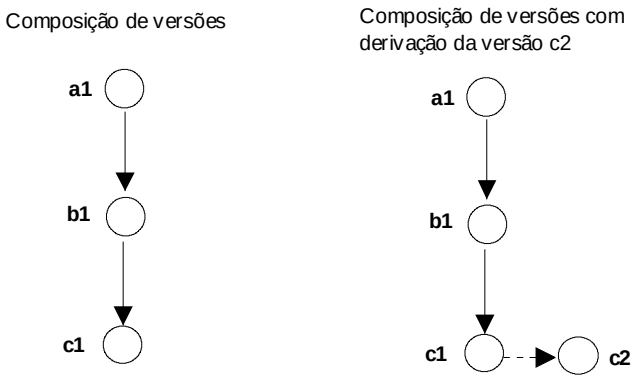

FIGURA 2.2 − Escopo da Notificação

Foram analisadas cinco propostas que utilizam o mecanismo de notificação ou propagação de mudanças. Nesta seção, é apresentada cada proposta e as características acerca deste mecanismo, seguindo a ordem cronológica de publicação.

2.1.1 Proposta segundo ATWOOD

Esta proposta aborda apenas o mecanismo de propagação e não apresenta nenhum mecanismo de notificação.

A técnica de propagação chamada *version percolation* é proposta por Atwood [ATW 85]. Segundo essa técnica, quando uma nova versão é derivada, o sistema automaticamente gera novas versões de todos os objetos que fazem referência direta a essa nova versão. A figura 2.3 mostra um exemplo usando esta técnica. Quando novas versões de b1 e c1 são derivadas, as versões a2, a3 e a4 são criadas automaticamente.

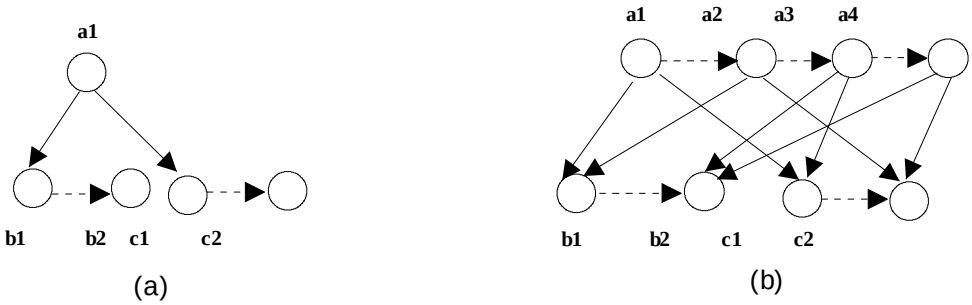

FIGURA 2.3 − Técnica *Version Percolation*

A técnica *version percolation* apresenta alguns problemas, tais como:

- um grande número de versões desnecessárias podem ser geradas, já que todas as combinações possíveis são criadas. Por exemplo, se o usuário desejar que somente uma nova versão de a1 seja criada contendo referências às novas versões b2 e c2, isso não será possível com a utilização dessa técnica;

- no caso da exclusão, o usuário não é notificado quando uma versão é excluída.

2.1.2 Proposta segundo CHOU e KIM

Um modelo de controle de versões para sistemas CAD e CAM foi desenvolvido por Chou e Kim [CHO 86, CHO 88] no qual foram exploradas questões operacionais e semânticas, incluindo identificação de versões, ligações estáticas e dinâmicas entre versões, versões de esquema e notificação de mudanças. Este modelo foi, mais tarde, integrado ao modelo de dados do SGBD ORION.

Um mecanismo de notificação de mudanças foi proposto para oferecer aos sistemas de banco de dados a funcionalidade de reagir a mudanças de versões em um ambiente CAD. Esse mecanismo reage às seguintes ações:

- alteração de versões;
- exclusão de versões;
- criação de versões.

A notificação pode ocorrer através de duas técnicas: baseada em mensagens (*message−based*) ou baseada em sinais (*flag−based*). Na notificação baseada em mensagens, o sistema envia mensagens para notificar os usuários sobre as versões (dos objetos compostos) possivelmente afetadas.

Na notificação baseada em sinais, marcas são colocadas nas estruturas de dados armazenados. O usuário toma conhecimento de uma mudança quando explicitamente fizer acesso a uma versão de um objeto composto que ficou com as referências desatualizadas devido a modificações nas versões dos componentes utilizados. Cada versão de um objeto, tem dois selos temporais distintos: aprovação e notificação. O selo temporal de notificação de mudanças (CN) indica o tempo que a versão foi criada, ou a última vez que foi alterada, e o selo temporal de aprovação de mudanças (CA) indica a última vez que o projetista da versão aprovou as mudanças nesta. Para haver consistência dos dados, o CN deve ser menor ou igual ao CA.

Na técnica baseada em mensagem, o sistema envia mensagens para notificar usuários sobre as versões afetadas. Essa técnica pode ser classificada em imediata e postergada, dependendo se os usuários afetados são notificados imediatamente após as mudanças ou algum tempo mais tarde especificado pelo usuário. A notificação baseada em mensagem exige que cada versão V conserve uma lista de referências invertidas de versões que fazem referência a V e que requerem notificação das mudanças em V.

Para limitar o escopo da notificação, é definido, no modelo de Chou, que somente as versões que fazem referência diretamente às versões alteradas sejam notificadas.

2.1.3 Proposta segundo KATZ

O modelo proposto por Katz [KAT 87, KAT 90] tem o objetivo de prover uma terminologia e um conjunto de mecanismos para o controle de versões. Este modelo

gerencia unidades de projeto denominadas objetos, as quais correspondem aos arquivos em ambientes de projetos tradicionais e é implementado no sistema VERSION SERVER.

Dois problemas são apontados por Katz e devem ser resolvidos por um mecanismo de propagação de mudanças:

1. **proliferação de versões:** como limitar o escopo da propagação, pois raramente o projetista deseja criar novas versões de todos os objetos até a raiz da hierarquia de composição. Este problema é análogo ao da definição do escopo da notificação;

2. **ambigüidade:** como tornar não ambíguo o caminho usado para a propagação. A ambigüidade surge, por exemplo, quando uma mesma versão é referida por mais de uma versão do mesmo objeto composto.

Restrições de configurações como meio de limitar o escopo da propagação de mudanças são sugeridos por Katz. Para evitar a propagação ambígua são usados os mecanismos *check−in* e *check−out* em grupo. Nesse caso, o usuário define um subgrafo com as versões de interesse, a partir do qual, novas versões são criadas. Há casos nos quais *check−in* em grupo não garantem um resultado não ambíguo. Foram criados métodos que contém regras para resolver esse problema.

2.1.4 Proposta segundo MOURAD CHABANE OUSSALAH

A proposta descrita em [TAL 93, OUS 96, OUS 96a, OUS 97, URT 98] apresenta um modelo de versões que estende o modelo de dados do BDOO VERSANT com técnicas de controle de evolução de objetos complexos. Essa proposta faz parte de um projeto denominado LEOPARD, desenvolvido na Escola de Informática e Engenharia de Nîmes (EERIE), na França.

A proposta apresenta um mecanismo de propagação de mudanças que controla os seguintes eventos:

- criação de versões de instância;
- criação de versões de classe;
- exclusão de versões de instância;
- exclusão de versões de classe.

Para resolver o problema da criação de infinitas versões, o mecanismo [TAL93] oferece duas maneiras de marcar as versões que são avaliadas:

- **versão sensível**: a propagação é executada somente se o usuário determina que a versão é sensível à propagação;
- **atributo sensível**: a propagação é executada quando o atributo de ligação do objeto composto com o objeto componente é dito sensível à propagação.

Em ambos os casos, versão sensível e atributos compostos sensíveis, a propagação ocorre somente se a versão componente causadora da propagação encontra−se no estado permanente.

Em trabalhos mais recentes [OUS 96], o problema do escopo da notificação é resolvido através da opção *restricted* ou *extended,* as quais definem se somente as versões ligadas diretamente são atingidas pela propagação ou se o são todas as versões diretamente ou indiretamente ligadas a versão modificada.

Quando duas ou mais versões fazem referência a uma mesma versão componente e esta versão evolui, surge o problema da ambigüidade. Para resolver este problema, o autor define um atributo específico chamado *multipaths*, cujo valor pode ser uma versão ou um conjunto de versões. O projetista pode especificar quais versões são criadas. Isto só é permitido porque as estratégias de propagação estão ligadas às relações entre objetos.

O mecanismo de propagação disponibiliza três sentidos para a propagação: *forward* (somente as versões componentes da versão modificada são criadas), *backward* (somente as versões compostas que referenciam versões modificadas são criadas) e *mixed* (ambas as versões componentes e compostas que referenciam a versão modificada são criadas).

As opções de sentido e limitação de escopo são especificadas em regras vinculadas às referências entre objetos. Essas referências são representadas por relações de dependência (*dependence relations*).

O modelo de versões de Oussalah usa o formalismo ECA (Evento, condição e ação), no qual, reagindo a um evento E, se a condição C é verificada, a ação A é disparada. O mecanismo de propagação intercepta as operações *create−version( )* e *delete−version( )* e ativa as regras que foram definidas.

2.1.5 Proposta segundo Luiz Fernando Soares

A proposta descrita em [SOA 95] apresenta um modelo conceitual hipermídia com composições aninhadas, no qual uma composição de documentos pode conter outra composição recursivamente. Esse modelo, denominado NMC, é a base conceitual para o projeto hipermídia desenvolvido no departamento de Informática da Pontifícia Universidade Católica, no Rio de Janeiro. Um dos principais requisitos deste projeto é o trabalho cooperativo.

O mecanismo de propagação de mudanças proposto por Soares pode propagar as seguintes mudanças:

- alterações de versões;
- criação de versões.

Em [SOA 98], procedimentos são apresentados para evitar a proliferação inútil de versões de documento (denominado nó). Esses procedimentos são baseados nos conceitos de sessões de trabalho, nos estados de um nó, no esquema para criação de versões, atributos versionáveis e não versionáveis.

A solução, para evitar a proliferação de versões quando a mudança é alteração de versões, é definir os atributos como sendo versionáveis ou não versionáveis. Um atributo não versionável pode ter seu valor modificado sem a necessidade da criação de uma nova versão. Ao contrário, modificações no valor de um atributo versionável devem ser realizadas em uma nova versão do objeto, se este já estiver no estado permanente.

Para evitar a proliferação de versões quando a mudança é a criação de versões, é proposta a primitiva *check−in−one*, caso já exista uma representação do nó com o mesmo descritor, uma nova versão não é criada. Ao contrário da primitiva *check−in* que sempre cria uma nova versão, independente da existência de um nó com o mesmo descritor. Ambas as primitivas criam somente as versões ligadas diretamente à versão causadora da mudança, limitando assim o escopo da propagação.

A primitiva *open* permite criar uma versão de dados e recursivamente de cada componente em uma hierarquia de composição. Esta opção cria novas versões de nós que são direta e indiretamente ligadas à versão causadora da mudança, não limitando a expansão do escopo da propagação.

Esse modelo permite versões de elos (relacionamentos entre nós). Os problemas decorrentes da propagação de versões de elos e suas soluções ainda são tópicos de pesquisa neste projeto.

A tabela 2.1 apresenta um resumo dos aspectos referentes ao mecanismo de notificação e propagação de mudanças encontrados nas propostas analisadas nesta seção.

TABELA 2.1 − Comparação entre os Mecanismos Analisados

Propostas encontradas em Projetos de Pesquisa					
Características	**ATWOOD**	**CHOU E KIM**	**KATZ**	**OUSSALAH**	**SOARES**
Estratégia	Propagação	Notificação	Propagação	Propagação	Propagação
Solução para resolver o problema do escopo da notificação ou propagação	Não apresenta	Restrições: somente as versões diretamente ligadas às versões alteradas são modificadas	Restrições: de configurações	− Atributos: sensíveis e não sensíveis − Versões: sensíveis ou não sensíveis − Restrições: somente as versões com estado permanente propagam mudanças; − Atributos: modo e sentido	− Atributos: versionável e não versionável − Restricões: somente as versões ligadas diretamente a versão modificada são propagadas − Operação: *chek_in_one*
Solução para resolver o problema da ambigüidade	Não apresenta	*	Operações: *chek_in* e *chek−out* em grupo	Atributo: *multipath*	Não apresenta
Eventos que disparam o mecanismo	− criação de versões	− alteração de versões − exclusão de versões − criação de versões	− *chek−in* − *check−out*	− criação de versões de instância − criação de versões de classe − exclusão de versões de instância − exclusão de versões de classe	− alteração de versões − criação de versões

* não especificado na literatura consultada.

2.2 Propostas encontradas em SGBDOOs

Nesta seção, são analisados os serviços de notificação e propagação de mudanças existentes em diferentes sistemas. Uma comparação desses serviços é apresentada no final.

2.2.1 Itasca − Sistema de Gerenciamento de Banco de Dados para Objetos Distribuídos

O sistema ITASCA [IBE 99, SKI 99] está baseado em uma série de protótipos ORION. Esses protótipos iniciaram o seu desenvolvimento em 1985 na *Microelectronics and Computer Technology Corporation* (MCC) e no laboratório de sistemas distribuídos. A empresa IBEX adquiriu o sistema de gerência de banco de dados distribuídos ITASCA em 1995. A plataforma de execução deste sistema é UNIX.

O ITASCA estende a linguagem LISP com a programação orientada a objetos e recursos de banco de dados e fornece APIs para LISP, C e C++, denominadas, respectivamente, de LISP API, C API e C++ API.

Um serviço específico para suportar versionamento automático de objetos é fornecido pelo ITASCA. Somente as versões das classes declaradas como versionáveis podem ter versões de instâncias.

O mecanismo de notificação do ITASCA é integrado ao esquema do banco de dados: um ou mais atributos da classe podem ser declarados como notificáveis. Se uma classe tem pelo menos um atributo notificável, ela é chamada de classe notificável.

Uma característica desse mecanismo é que ele não é suportado pela evolução de esquemas, isto é, a classe deve ser declarada como tendo atributos notificáveis quando é criada.

Uma característica ausente nesse mecanismo é a possibilidade de ter um objeto notificado e outro não notificado na mesma classe.

No momento em que uma classe é criada, se um atributo é declarado como notificável, o sistema incorpora à classe um novo atributo chamado *timestamp*. Esse atributo guarda o instante em que o objeto foi alterado.

ITASCA dispõe de dois tipos de notificação: notificação passiva e notificação ativa. Cada objeto mantém duas listas de eventos notificáveis, uma para cada tipo da notificação. Os usuários podem determinar o conjunto de eventos que são notificáveis através das mensagens: *(set−passive−notification object ´(ListOfEvents))* e *(set−active−notification object ´(ListOfEvents))*.

A notificação passiva, também chamada notificação por sinal, é usada para permitir a verificação da consistência dos dados pelos usuários, principalmente a integridade referencial. Os tipos de eventos notificáveis por este mecanismo são exclusão e alteração de objetos. Todos os usuários que possuem acesso ao objeto podem consultar as notificações. O usuário pode verificar a consistência dos dados e então aprovar a mudança. O sistema fornece um atributo de notificação chamado *approval−*

timestamp. Este atributo guarda o instante de tempo em que o usuário "aprovou" a ocorrência do evento.

A figura 2.4 ilustra um exemplo que gera uma referência inválida. O objeto m1 possui uma referência composta dependente do objeto a2. O objeto a1 também referencia o objeto m1 por outro tipo de referência composta, não dependente. Se o objeto a2 for excluído, o objeto m1 é excluído automaticamente e com isso o objeto a1 ficará com referência inválida. O mecanismo de notificação não impede que um objeto adquira uma referência inválida mas disponibiliza a informação para o usuário.

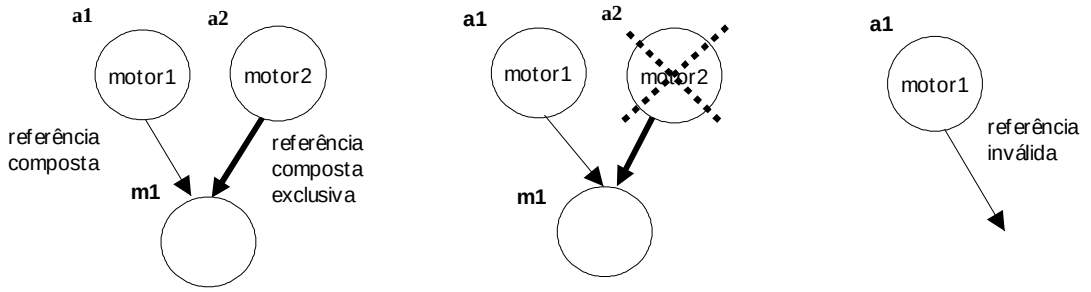

FIGURA 2.4 − Criando uma Referência Inválida

Outro tipo de notificação encontrado no ITASCA é a notificação ativa ou notificação baseada em *daemon.* A notificação ativa é útil quando há necessidade da divulgação da informação imediatamente ou quando mudanças no banco de dados são necessárias após a ocorrência de um evento. Os eventos que podem ativar este tipo de mecanismo são: exclusão de objetos, alteração, criação de uma nova versão, *checkin* e *checkout*. Por *default*, quando estes eventos ocorrem é enviada uma mensagem para o usuário descrevendo a mudança e o usuário que a causou. Pode−se também determinar a execução de outros métodos, além dos métodos que enviam mensagens.

O usuário deve especificar quais mudanças e quais objetos são de seu interesse. A lista de eventos de cada objeto para a notificação ativa consiste de pares compostos de evento e nome do usuário. Se, por exemplo, o usuário "fonseca" quer ser notificado quando o objeto *m1* é alterado, este deve utilizar a seguinte operação: *(set−active−notification m1 '(:update))*. A lista de notificação ativa do objeto *m1* contém o par *(:update ('fonseca')nill)*. As operações *find−users−to−notify* e *send−notification−mail* são usadas respectivamente para encontrar os usuários que devem ser notificados e para enviar a mensagem a eles.

Um exemplo da utilização de notificação pode ser encontrado em [ALL 97], onde o mecanismo de notificação ativa do Itasca é usado para implementação de um serviço *multicast*. Esse serviço é responsável pela comunicação dos agentes no *ItascaFlow* (tecnologia de banco de dados objeto para o sistema *workflow − Adaptive Workflow Systems*).

2.2.2 ObjectStore − Sistema de Gerenciamento de Banco de Dados Orientado a objetos

O ObjectStore [OBJ 99, LAU 98], distribuído pela *Object Design*, é um sistema de gerenciamento de banco de dados orientado a objetos para aplicações de alto desempenho, Internet e outros ambientes de computação distribuída. O sistema apresenta como características principais: a segurança, manuseio de grandes coleções, alta velocidade e suporte a transações complexas.

O modelo de dados deste sistema está baseado nos princípios fundamentais de orientação a objetos, como o conceito de identidade de objetos, classes, composição, herança, funções virtuais, *templates* e encapsulamento. Como utiliza as linguagens Java e C++, a característica de persistência aparece como uma extensão destas linguagens.

Este sistema possui um serviço de notificação [ROS 99] que permite a um cliente ObjectStore notificar outros clientes assim que um determinado evento ocorrer. Um evento pode ser, por exemplo, uma alteração de um determinado objeto no banco de dados. A funcionalidade de notificação do ObjectStore está disponível nas APIs Java e C++.

Uma notificação, para o ObjectStore, é um objeto transiente Java. O fluxo de uma notificação neste sistema é descrito a seguir. Os métodos citados estão disponíveis na API Java [OBJ 99a]:

1. Uma aplicação gera uma notificação no ObjectStore através de uma chamada ao método *new Notification(Object location, int kind, String message)*. O parâmetro *location* especifica um objeto persistente, enquanto os parâmetros *kind* e *message* provêm informações sobre o evento ocorrido.

2. Cada sessão aberta do ObjectStore mantém, no gerenciador de *cache*, uma lista de notificações. Para subscrever notificações nesta lista, os métodos *Notification.subscribe(Placement placement)* ou *Notification.subscribe (Object location)* são utilizados. O parâmetro *placement* pode especificar um banco de dados ou um segmento e o parâmetro *location* especifica um objeto persistente. Se notificações são subscritas para um determinado segmento ou banco de dados, a sessão subscrita recebe qualquer notificação referente aos objetos pertencentes a eles.

3. Um evento envolvendo um objeto, segmento ou banco de dados ocorre e a aplicação envia uma notificação para o servidor ObjectStore. O método *public void notifyImmediate( )* é utilizado para enviar uma notificação ou um conjunto de notificações imediatamente após a ocorrência de um evento e o método *public void notifyOnCommit( )* para enviar uma notificação somente quando a transação é encerrada. Transações são independentes de notificações e subscrições, isto é, uma transação não é desfeita se a notificação relacionada não for criada ou subscrita.

4. O servidor recebe uma notificação, verifica quais sessões estão subscritas e assincronamente envia as notificações para o gerenciador de *cache* de cada sessão recebedora. Uma sessão inicializa uma *thread* com o objetivo único

de receber notificações. Esta *thread* chama o método *Notification.receive( )* passando como argumento o tempo de espera. Uma sessão recebe uma notificação e executa a ação especificada na aplicação. A ação pode ser, por exemplo, o envio de uma mensagem ou a alteração de estruturas dos dados transientes da aplicação.

Uma sessão pode cancelar subscrições, imediatamente, através do método *Notification.unsubscribe( )*. Ela pode cancelar subscrições para uma notificação particular, mas ainda receber notificações para um objeto, segmento ou banco de dados que já haviam sido colocados na lista de notificações.

O mecanismo do sistema OBJECTSTORE é de baixo nível, sendo necessária a implementação do mecanismo de notificação na aplicação, utilizando a API.

2.2.3 O2 − Sistema de Gerência de Banco de Dados Orientado a Objetos

O sistema de banco de dados Objeto O2, distribuído pela *Ardent Software* [ARD 99], é um sistema aberto, de alto desempenho, voltado para aplicações da indústria. O O2 suporta um sofisticado mecanismo de versionamento [O2T 96], o qual permite o gerenciamento de versões e configurações de objetos.

O sistema O2 possui um serviço de notificação [ARD 99a] avançado para aplicações dirigidas pelo evento chamado *O2Notification.* Esse serviço é baseado no modelo *Publish and Subscribe.* A etapa *Publish* ocorre quando uma aplicação cliente registra um tipo de evento no servidor O2. O cliente registra−se no servidor, especificando os eventos para os quais deseja ser notificado. O servidor monitora as atividades relacionadas aos eventos registrados e notifica os clientes quando estes ocorrem. O servidor O2 mantém o armazenamento persistente das informações sobre clientes e registro de eventos.

O serviço de notificação do O2 é utilizado quando:

- objetos e ou versões de banco de dados são alterados ou excluídos,
- clientes de banco de dados conectam−se ou desconectam−se no banco,
- um evento definido pelo usuário ocorre.

A figura 2.5 mostra o fluxo da notificação no sistema O2.

A notificação de eventos definidos pelo usuário é disparada somente pela aplicação, enquanto as alterações nos objetos e eventos de conexão de cliente são detectados e acionados automaticamente pelo sistema de banco de dados. O servidor suporta notificações múltiplas por cliente. Clientes podem especificar filtros para limitar o escopo de eventos. A ocorrência de eventos é assíncrona a sua divulgação.

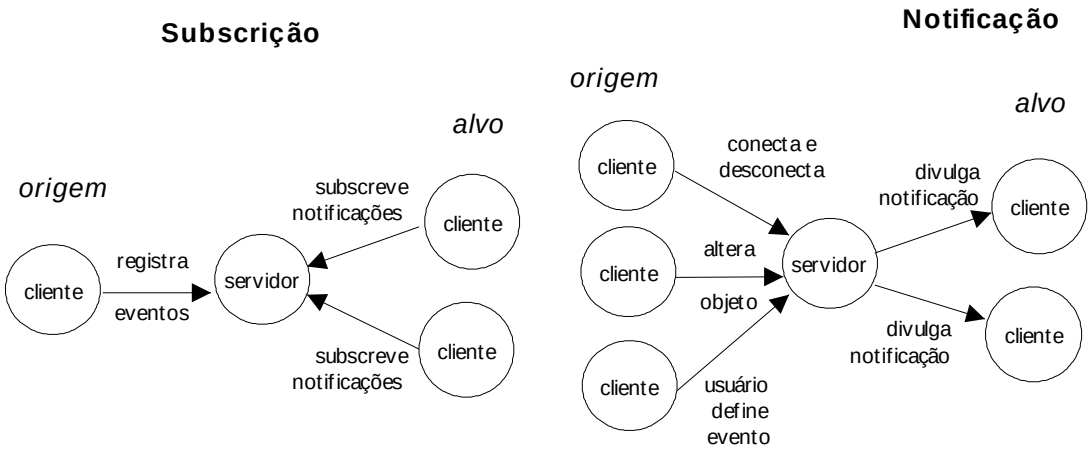

FIGURA 2.5 − Mecanismo de Notificação do Sistema O2

As informações fornecidas aos clientes notificados incluem:

- **identificador de objeto:** em eventos de alteração de objetos;
- **nome do cliente:** em eventos de conexão no banco;
- **nome fornecido pela aplicação:** em eventos definidos pelo usuário.

Informações estatísticas sobre os eventos podem ser consultadas no servidor O2 a qualquer momento.

2.2.4 GemStone − Sistema de Gerência de Banco de Dados Orientado a Objetos

O sistema GemStone [LAU 98, FUR 98] foi desenvolvido pela *Servio Logic Development Corporation*, em 1987, mudando seu nome mais tarde para G*emStone Systems Inc*. Esse sistema combina os aspectos de uma linguagem orientada a objetos, no caso *Smalltalk*, com a funcionalidade de um sistema de banco de dados.

O GemStone [SKI 99] possui um mecanismo de notificação de mudanças que permite a uma aplicação ser notificada quando mudanças ocorrem no banco de dados. Esse mecanismo apresenta duas abordagens:

1. mecanismo de notificação;
2. mecanismo sinalizador de troca.

Para cada objeto modificado, uma aplicação pode receber um relatório que contém as seguinte informações:

- OID do objeto;
- tipo de mudança;
- usuário que realizou a mudança.

O mecanismo de notificação permite que várias aplicações sejam notificadas de qualquer mudança no banco de dados. Cada aplicação deve declarar quais objetos são monitorados, sendo o conjunto destes objetos denominado *Notify set* da aplicação. O processo *Stone* guarda o endereço do *Notify set* da aplicação e envia uma mensagem a eles, no caso de um objeto que pertence ao *Notify set* ser modificado. As mensagens são tratadas pela aplicação como mensagens de erro e podem ser recebidas durante a chamada a uma função GCI (GemStone *C Interface*) pela aplicação que está sendo executada. A aplicação pode verificar explicitamente se mudanças ocorreram no *Notify set*, usando funções especiais.

O mecanismo de notificação do GemStone é assíncrono e possui duas desvantagens:

- modificações que não interessam causam notificações;
- a aplicação pode receber informações de quais objetos foram modificados, mas não recebem informações como: tipo de mudança e quem as realizou.

O segundo tipo de abordagem que o GemStone provê é o mecanismo sinalizador de troca (*exchange signal mechanism*). As mensagens são tratadas pela aplicação recebedora como mensagens de erro, conseqüentemente não podem ser recebidas sincronamente, mas somente quando uma função GCI é chamada. Se o conteúdo do banco de dados for alterado de outra maneira, este mecanismo não é executado. Cada mensagem recebida possui um número de identificação, a sessão que enviou a notificação e opcionalmente uma *string* que contém uma descrição ou qualquer informação necessária.

Se uma aplicação realiza uma modificação usando o mecanismo sinalizador de troca, pode enviar a outra aplicação o tipo de modificação realizada, mas não o OID do objeto que foi alterado. No mecanismo sinalizador, a aplicação que causou a mudança é responsável pela notificação de outras aplicações.

A tabelas 2.2(a), 2.2(b) e 2.2(c) apresentam um resumo dos aspectos referentes ao mecanismo de notificação e propagação de mudanças encontrados nos sistemas analisados nesta seção.

TABELA 2.2 (a) − Comparação entre os Mecanismos Analisados

SISTEMAS DE GERENCIAMENTO DE BANCO DE DADOS				
Características	**ITASCA**	**OBJECTSTORE**	**O2**	**GEMSTONE**
Tipos de estatrégias	− notificação passiva − notificação ativa	notificação ativa	notificação ativa	notificação ativa (mecanismo de notificação e mecanismo sinalizador de troca)
Trata as mudanças ocorridas	− em objetos − em versões	− em objetos − em segmentos	− em objetos − em versões	em objetos
Eventos que acionam o mecanismo	− alteração de objetos − exclusão de objetos − criação de objetos − *check−in* e *check−out* de objetos − alteração de versões − exclusão de versões − criação de versões − *check−in* e *check−out* de versões	− alteração de objetos − exclusão de objetos − inicialização de uma sessão do banco de dados − finalização de uma sessão de banco de dados	− alteração de objetos − exclusão de objetos − alteração de versões − exclusão de versões − inicialização de uma sessão do banco de dados − finalização de uma sessão de banco de dados − ocorrência de um evento definindo pelo usuário	*

* não especificado na literatura consultada.

TABELA 2.2 (b) − Comparação entre os Mecanismos Analisados − Continuação

SISTEMAS DE GERENCIAMENTO DE BANCO DE DADOS				
Características	**ITASCA**	**OBJECTSTORE**	**O2**	**GEMSTONE**
Ações executadas pelo mecanismo como resposta à eventos	− uma mensagem é enviada ao usuário ou uma ação definida pelo usuário é executada (notificação ativa) − uma lista de mudanças é mantida pelo sistema (notificação passiva)	implementadas na aplicação	implementadas na aplicação	implementadas na aplicação
Acrescenta atributo a classe notificável	acrescenta *timestamp*	*	*	*
Implementação do mecanismo	funções definidas em LISP	métodos definidos nas API C++ e API Java	− módulo *O2 Notification* − não especifica a linguagem, se é: *C++, java ou O2C*	GCI (GEMSTONE C Interface)

* não especificado na literatura consultada.

TABELA 2.2 (c) − Comparação entre os Mecanismos Analisados − Continuação

SISTEMAS DE GERENCIAMENTO DE BANCO DE DADOS				
Características	**ITASCA**	**OBJECTSTORE**	**O2**	**GEMSTONE**
Armazenamento das Notificações	− as notificações são persistentes até que o usuário "aprove" a mudança (notificação passiva) − não é persistente (notificação ativa)	não são persistentes	são armazenadas no servidor	não são persistentes
Armazenamento das subscrições	*	não são persistentes	são armazenadas no servidor e não são persistentes	*

* não especificado na literatura consultada.

2.3 Proposta segundo GAMMA

O padrão de projeto comportamental PUBLISHER−SUBSCRIBER [BUS 96], também conhecido como OBSERVER [GAM 99], tem como objetivo manter sincronizado o estado dos objetos cooperativos. Esse padrão define uma dependência um−para−muitos entre objetos, de modo que todos os objetos dependentes (denominados *observers*) sejam notificados e alterados automaticamente quando um determinado objeto (denominado *subject*) tiver seu estado alterado. O OBSERVER é utilizado para modelar o comportamento de objetos após a ocorrência de mudanças.

Algumas ferramentas para interfaces de usuário empregam o padrão OBSERVER [GAM 99]. O exemplo mais conhecido aparece na linguagem *Smalltalk*, que utiliza este padrão no *framework Model/View/Controller* (MVC). No *framework*, responsável pela interface do usuário, a classe *Model* exerce o papel do *subject*, enquanto a classe *View* exerce o papel do *observer*.

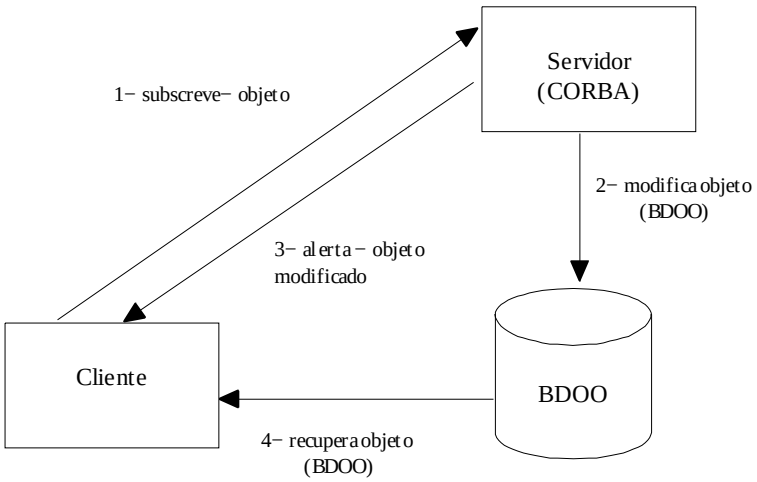

FIGURA 2.6− *Observer* (notificação distribuída) extraída de [GAR 99]

Outro uso deste padrão aparece no mecanismo de notificação de eventos disponível no sistema IRIDIUM [GAR 99]. Esse mecanismo permite a um cliente de banco de dados receber alertas quando um determinado objeto tem o seu estado alterado, conforme ilustra a figura 2.6. A notificação é realizada pelo componente SCS (*System Control Segment*) do IRIDIUM .

2.4 Conclusões

Apesar de reconhecida a importância dos mecanismos para auxiliar a evolução de dados em modelos e em sistemas de banco de dados [CHO 86, KAT 90, ARD 99a, OBJ 99a], ainda não existe um consenso entre propostas apresentadas, nem um mecanismo completamente definido ou totalmente implementado. Um mecanismo com suporte às necessidades de evolução é um requisito importante para aplicações de banco de dados não convencionais.

As diferenças entre os mecanismos analisados dizem respeito basicamente aos eventos monitorados, às estratégias utilizadas e à forma de resolver os problemas de ambigüidade e do escopo de notificação e propagação.

Soluções para resolver o problema do escopo de notificação e propagação e o problema da ambigüidade são apresentadas através de restrições e incorporação de atributos, cláusulas e operações ao modelo de dados.

As estratégias utilizadas permitem especificar quais ações são executadas em resposta aos eventos executados.

Cabe salientar ainda que, além dos SGBDs destacados neste trabalho, outros sistemas apresentam algum mecanismo de notificação ou propagação, como por exemplo JADE, GODDS e VERSANT [LIN 99, VER 99, VER 99a]. Esses sistemas não apresentam características distintas em relação aos demais aqui analisados, por isso não constam neste trabalho.

Outro mecanismo de notificação, não apresentado neste capítulo, adotado pela proposta de Noronha [NOR 98] (versões para documentos) é uma combinação das técnicas baseadas em mensagens e em sinais. No enfoque adotado por Noronha, o envio de mensagens é imediato e somente os documentos compostos que fazem referência direta à versão modificada são notificados. O processo de notificação é operacional apenas para as versões com referências estáticas para os componentes cujas versões foram modificadas. São consideradas modificações somente as seguintes operações: promoção de uma nova versão componente para o *status* estável e passagem de uma versão componente para o *status* obsoleto.

3 O modelo de versões de Golendziner

Neste capítulo é descrito, sucintamente, o modelo de versões proposto por Golendziner em [GOL 95, GOL 95a, GOL 95b, GOL 95c, GOL 97]. Seu desenvolvimento estende a um modelo de dados orientados a objetos conceitos e mecanismos que suportam a definição de objetos, versões e configurações.

3.1 Objeto, Objeto Versionado e Versão

Versão é uma descrição de um objeto num determinado momento do tempo, ou sob um certo ponto de vista, cujo registro é importante para a aplicação. Em um modelo orientado a objetos, uma versão é um objeto de primeira classe, possui um identificador próprio, podendo ser diretamente manipulada ou consultada, como qualquer outro objeto do sistema.

As versões de uma entidade do mundo real ficam agrupadas e constituem um objeto versionado (figura 3.1). Um objeto versionado também é um objeto de primeira classe e contém informações sobre as versões associadas, além de possuir propriedades próprias. Outra característica do objeto versionado é que os atributos possuem valores comuns a todas as versões. Cada versão pertence exatamente a um objeto versionado.

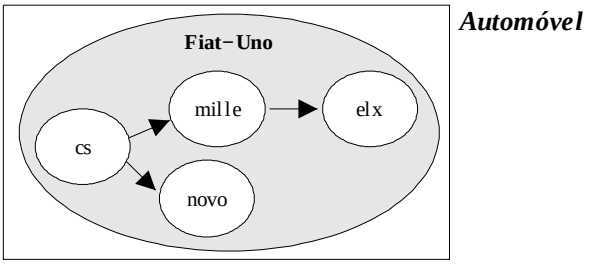

FIGURA 3.1 − Objeto versionado e suas versões

Os objetos (versionados ou não) que representam as mesmas propriedades e comportamento podem ser agrupados em classes. Como a propriedade de ser versionado pertence ao objeto, uma classe pode ter objetos versionados e não versionados como instâncias.

Cada objeto versionado possui uma versão que é considerada sua versão corrente, sendo automaticamente mantida pelo sistema como a mais recentemente criada. O usuário pode especificar uma versão diferente como a corrente e, neste caso, a versão ficará fixa, não mudando com o surgimento de novas versões. A versão corrente é utilizada sempre que o usuário solicitar uma operação sobre um objeto versionado sem especificar uma de suas versões.

3.2 Identificadores de Objetos

Cada identidade do mundo real é modelada como um (ou mais) objeto(s) que possui um identificador definido pelo sistema (*Object Identifier* − OID), independente do seu valor. Cada objeto tem um estado e um comportamento. O estado de um objeto é dado a qualquer momento pelos valores de suas propriedades (atributos). Os valores de propriedades podem ser primitivos, tais como inteiros, seqüência de caracteres, booleanos, ou não primitivos. O comportamento de um objeto é especificado pelas operações (métodos) que manipulam propriedades de um objeto.

Um objeto versionado possui número de versão nulo e um não versionado possui número de versão igual a 1. Assim, um objeto não versionado é identificado da mesma maneira que a primeira versão de um versionado. A evolução de um objeto não versionado para versionado provoca a criação de um objeto versionado, não influindo na identificação do objeto existente, que passa a ser a primeira versão. Referências que existiam para o objeto não versionado passam a ser referências estáticas para a primeira versão do novo objeto versionado, não impactando o usuário.

A estrutura proposta para o identificador é a seguinte:

OID = <id entidade, classe, número da versão>

Exemplificando uso do identificador do objeto versionado Fiat-Uno, ilustrado na figura 3.1, o OID corresponde à <fiat−uno, Automovel, nulo>.

3.3 Estado das Versões

Versões são classificadas para refletir o seu estágio de desenvolvimento e/ou consistência. Dependendo do estado da versão, certas operações não são possíveis.

Os estados que uma versão pode assumir no modelo de versões são:

- **trabalho**: se uma versão encontra−se nesse estado, pode ser alterada ou removida. Uma versão recém criada com a operação de derivação possui o estado em trabalho.
- **estável** : se uma versão encontra−se nesse estado, pode ser excluída se não existirem objetos que a referenciam. Uma versão neste estado não pode ser alterada. Se uma versão está no estado em trabalho e serve como base para a derivação, ela é promovida automaticamente para o estado estável.
- **consolidada**: se uma versão encontra−se nesse estado ela não pode ser alterada e nem removida. Uma versão pode tornar−se consolidada por uma operação de promoção.

3.4 Composição e Objetos Complexos

Também chamado de objeto complexo, pode ser definido como objeto composto de outros objetos, constituindo uma hierarquia. A composição de objetos é expressa pela inclusão de um identificador (OID) do objeto (componente) como valor

de atributo de outro objeto (composto).

Como objetos compostos também podem possuir versões, cada versão de um objeto composto deve ter suas ligações estabelecidas com os objetos componentes. A ligação entre uma versão do objeto composto e uma versão de cada um de seus objetos componentes chama−se configuração.

Objetos que estão sendo referidos por outros não podem ser excluídos.

3.5 Configurações

Considerando um objeto composto, em que os componentes podem ser versionados, uma configuração associa exatamente uma versão para cada um desses componentes. Como o modelo apresenta herança por extensão, esse processo deve incluir a definição de uma versão para cada um dos níveis onde o objeto está representado. Assim, diferentes escolhas de versões para componentes e/ou ascendentes geram diferentes configurações para o mesmo objeto, o que permite considerar que uma configuração é uma versão especial de um objeto, chamada versão configurada.

3.6 Referências Estáticas e Dinâmicas

Quando objetos que apresentam versões são usados como componentes de outro, podem ser estabelecidos dois tipos de referências: referência estática ou referência dinâmica.

Uma referência estática é uma referência simples a uma versão do objeto, ao passo que uma referência dinâmica significa que uma versão específica será escolhida em tempo de execução, quando o objeto for recuperado. Num ambiente de projeto, uma referência dinâmica indica que o usuário ainda não sabe qual versão do componente será usada, ou que ele quer deixar em aberto a escolha para testar diferentes opções.

3.7 Herança por Extensão

O tipo de herança proposto neste modelo é herança por extensão. Neste tipo de herança versões são admitidas em vários níveis da hierarquia de herança. Desta forma, a modelagem de entidades do mundo real pode ser feita em vários níveis de abstração, projetando ou modificando características de um objeto em um nível de cada vez. Por exemplo, se a classe *Automovel* é definida como extensão da classe *Veiculo*, cada objeto *automovel* possui um objeto *veiculo* como seu ascendente (figura 3.2). Cada versão deve possuir pelo menos um ascendente que lhe corresponda, isto é, quando uma versão é criada, obrigatoriamente deve ser ligada a um ascendente. Pode ocorrer que uma versão tenha mais de um ascendente, significando que as mesmas características definidas naquele nível podem ser ligadas a diferentes características no nível superior da hierarquia de herança. Por outro lado, uma mesma versão em uma *superclasse* pode corresponder a mais de uma versão na subclasse.

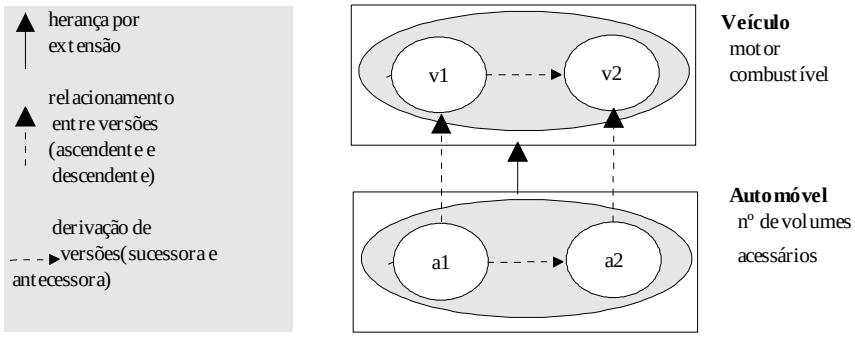

FIGURA 3.2 − Herança por extensão

Ficam assim estabelecidas correspondências (mapeamentos) entre versões de seu objeto ascendente na superclasse. A correspondência estabelece uma restrição de integridade, que é especificada quando da definição do relacionamento de herança entre uma classe e sua superclasse (definição de esquema), e deve ser mantida pelo sistema.

A correspondência definida entre as versões em uma classe e aquelas em sua superclasse pode ser do tipo n:m, 1:1, 1:n ou n:1.

3.8 Conclusões

Este capítulo apresentou um resumo do modelo que estende o modelo orientado a objetos com versões proposto em [GOL 95], descrevendo aspectos que são indispensáveis para o entendimento desta dissertação.

As operações que podem ser feitas sobre as versões, tais como: criação, modificação e remoção não foram descritas neste capítulo, porque serão apresentadas no capítulo 4, onde o mecanismo proposto é descrito em detalhes.

O modelo de Golendziner é consolidado e reconhecido pela comunidade acadêmica desde que foi proposto em 95. Atualmente ainda é alvo de muitos estudos como, por exemplo, mecanismos que tratam modificações em banco de dados estão sendo incorporados a este modelo. As modificações tratadas são as realizadas em dados do domínio do problema [FON 98] ou em esquemas (evolução de esquemas) [GAL 98, ROM 00].

No capítulo seguinte, será apresentado um mecanismo de notificação e propagação para tratar as mudanças ocorridas em objetos que não são avaliadas pelo modelo de versões de Golendziner.

4 Mecanismo proposto

Em aplicações não convencionais como, por exemplo, de projeto e de documentos hipermídia, existe a necessidade de propagação de mudanças e divulgação de informações sobre eventos ocorridos nos dados [BOR 95, OUS 97, SOA 98]. Essas aplicações possuem uma maior necessidade de suporte a cooperação entre usuários, já que estes trabalham em grupo.

Neste capítulo, é apresentado um mecanismo de notificação e propagação de mudanças para o modelo de versões proposto por Golendziner [GOL 95]. Esse mecanismo visa a tratar a evolução de dados e auxiliar em trabalhos cooperativos, não sendo avaliada a evolução de esquema[1].

O mecanismo sugerido como trabalho futuro por Golendziner, em [GOL 95], visava a tratar somente da evolução de versões de objetos componentes. O presente trabalho, além de considerar este tipo de mudança, também se propõe a tratar das mudanças ocorridas em objetos versionados e não versionados.

Quando os dados evoluem, por exemplo, quando uma nova versão de objeto é criada, algumas ações são desejáveis. Este trabalho apresenta possibilidades para executá−las automaticamente, propagando mudanças e divulgando informações sobre estas. Com isso, a consistência dos dados é mantida e as informações sobre as mudanças ocorridas são divulgadas.

O suporte à interação entre usuários, outro aspecto relevante para este trabalho, é um requisito importante para auxiliar em trabalhos cooperativos, podendo ser fornecido de duas formas, segundo [LIM 96, TOL 99]. Na primeira, a interação entre usuários é realizada através de um espaço de trabalho especial. Cada projetista tem seu espaço privado mas pode trocar informações com outros projetistas através de um espaço de grupo. Nesse espaço de grupo, itens de dados compartilhados podem ser armazenados e recuperados através de protocolos de *checkin/checkout*.

Na segunda forma, a interação entre os usuários é realizada em tempo real. Uma ação de um usuário sobre um contexto compartilhado é imediatamente propagada para todos os membros do grupo. O modelo de versões não define espaços diferenciados de trabalho, sendo que, a segunda forma de interação entre usuários é provida através do mecanismo proposto.

Alguns problemas podem surgir quando se propagam mudanças. Um deles é a proliferação de versões, que envolve a questão de como limitar o escopo da propagação, pois raramente, o usuário deseja criar novas versões de todos os objetos até a raiz da hierarquia de composição. Esse problema é análogo ao da definição do escopo da notificação.

Outro problema apresentado pela propagação é a ambigüidade. A ambigüidade pode ocorrer quando uma mesma versão componente é referenciada por mais de uma versão do mesmo objeto composto.

1 Tratado em [GAL 98] e em [ROM 00].

No mecanismo proposto, três estratégias são estabelecidas: notificação ativa, notificação passiva e propagação. Essas estratégias são detalhadas na seção 4.1. As novas operações e estruturas adicionadas ao modelo de versões de Golendziner são detalhadas nas seções 4.2 e 4.3 .

A figura 4.1 ilustra a arquitetura desenvolvida neste trabalho, mostrando os relacionamentos existentes entre usuários, banco de dados e o mecanismo de notificação e propagação de mudanças proposto. Os usuários subscrevem em listas os objetos que precisam ser monitorados. O mecanismo é responsável pelas ações que devem ser disparadas logo após a ocorrência de determinados eventos. Essas ações podem ser:

- envio de uma mensagem para os usuários;
- propagação de mudanças nos dados do banco de dados;
- armazenamento persistente de notificações para posterior consulta.

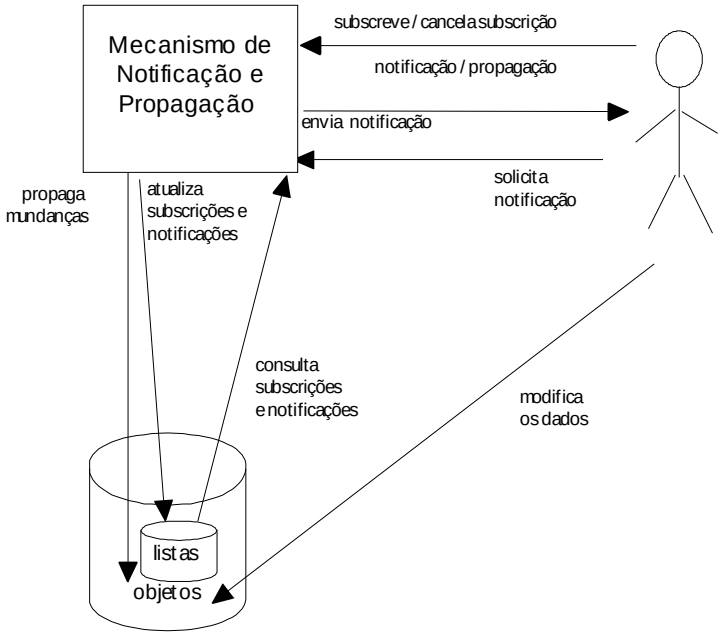

FIGURA 4.1 − Mecanismo de notificação e propagação

Para subscrever objetos nas listas de subscrições e para verificar a consistência dos dados, o usuário deve abrir uma seção do banco de dados. Além disso, ele deve ter permissão para modificar os objetos para conseguir subscrevê−los. Sendo assim, o mecanismo não oferece risco adicional à segurança dos dados. Isso deve ser garantido pela implementação.

O mecanismo reage aos eventos somente após a confirmação destes. Conforme mostra a figura 4.2, uma falha pode ocorrer depois que a transação 1 é encerrada. Neste caso, deve ser definido um procedimento para garantir o processamento completo da notificação, mesmo após a ocorrência de falhas. Isso também é válido quando a estratégia escolhida é a propagação.

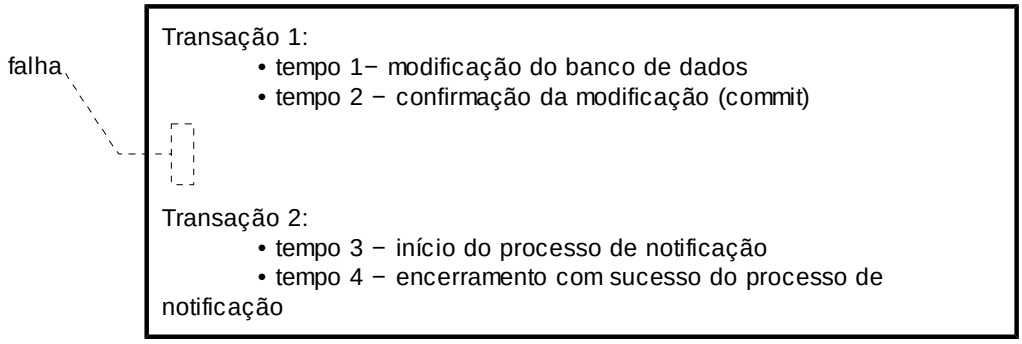

FIGURA 4.2 − Ocorrência de Falha

4.1 Estratégias

Visando atender às diferentes necessidades dos usuários, três estratégias são estabelecidas: notificação passiva, notificação ativa e propagação. O mecanismo reage de maneira distinta às mesmas ações, dependendo da estratégia escolhida. Por exemplo, se uma nova versão de um objeto é criada e a estratégia escolhida é a notificação ativa, uma mensagem é enviada ao usuário. Por outro lado, se a estratégia escolhida é a propagação, uma nova versão do objeto composto que faz referência à versão modificada é criada.

4.1.1 Notificação Passiva e Notificação Ativa

A notificação passiva é usada para auxiliar no controle da consistência dos dados. O usuário pode verificar se mudanças ocorridas no conteúdo do banco de dados não tornaram os dados inconsistentes, utilizando a operação *consult_notification* definida neste trabalho. Nessa estratégia, em nenhum momento, o usuário é forçado a receber notificações.

Quando um evento previamente subscrito para a notificação passiva ocorre, o armazenamento das informações referentes a este evento é realizado imediatamente após a ocorrência deste. Essas mensagens são persistentes e são estruturadas de forma que os usuários possam consultar mais tarde. Quando as informações se tornam desnecessárias ao usuário, a operação *delete_notification* para a exclusão de notificações é especificada.

A notificação ativa é utilizada quando o usuário necessita de informação em tempo real. Nessa estratégia, mensagens são enviadas ao usuário do banco de dados com informações sobre as mudanças ocorridas no conteúdo deste. Entretanto, essas mensagens não são persistentes, ou seja, são armazenadas temporariamente até que o envio seja realizado com sucesso.

Uma mensagem pode ser enviada de duas maneiras: através de uma mensagem do sistema operacional e através de correio eletrônico.

No enfoque adotado pelo mecanismo, o envio de mensagens é imediato e somente os usuários que subscreveram os objetos modificados, recebem a notificação. Com a utilização do modelo *publish and subscribe* [VER 99A], somente os objetos subscritos são monitorados. Com isso, o problema do escopo da notificação é resolvido.

A tabela 4.1 relaciona os eventos correspondentes às operações definidas no modelo de versões, à descrição destas operações e à classe a qual elas pertencem. As classes e operações são apresentadas detalhadamente em [GOL 95].

TABELA 4.1 − Eventos Controlados pela Notificação Passiva e pela Notificação Ativa

Evento	Operação definida no modelo de versões	Descrição	Classe
modify	**modify**(OID, list (atributo:[+|−]valor))	alteração do valor de uma propriedade	*Object*
modify (nome do atributo)	**modify**(OID, list (atributo:[+|−]valor))	alteração do valor de uma propriedade	*Object*
change	**change_current**(OID objeto versionado [,OID versão])	alteração da versão corrente	*VersionedObject*
promote	**promote**(OID[,**all ascendants**] [,**all referenced**])	alteração do *status* de uma versão	*VersionedObject*
derive	**derive_version**(set(OID) [, **ascendant**:[classe1:] set (OID) [, [classe2:] set(OID)]*] [,**descendant**:[classe3:] set(OID) [,[classe4:] set (OID)]*]:OID	criação de uma nova versão	*Object*
delete	**delete_object**(OID)	exclusão de objetos, versões e objetos versionados	*Object*

4.1.2 Propagação

Em uma composição, existem objetos compostos, utilizando como componentes objetos que representam versões. Modificações nos componentes, como a remoção ou surgimento de uma nova versão, podem causar alterações nos objetos que os referenciam. Com isso surge a necessidade de propagar automaticamente essas modificações para os objetos compostos.

TABELA 4.3 − Eventos Controlados pela Propagação

Evento	Operação definida no Modelo de Versões	Descrição	Classe
derive	**derive_version**(set(OID) [, **ascendant**:[classe1:] set (OID) [, [classe2:]set(OID)]*] [,**descendant**:[classe3:] set(OID) [,[classe4:] set (OID)]*]:OID	criação de uma nova versão	*Object*
delete	**delete_object**(OID)	exclusão de versão ou objeto	*Object*

Quando a estratégia escolhida é a propagação, os eventos controlados dizem respeito à evolução de versões e exclusão de objetos. Os eventos monitorados são: *delete* e *derive*, conforme mostra a tabela 4.3.

A propagação, dependendo de que tipo de objeto se trata o observador, pode assumir os seguintes sentidos:

- ***backward:*** a versão composta que referencia a versão modificada é atingida. Nesse caso o observador é a versão composta*;*
- ***forward:*** a versão componente da versão modificada é atingida. Nesse caso o observador é a versão componente;
- ***mixed:*** a versão componente e a versão composta que referenciam a versão modificada são atingidas. Nesse caso, existem pelo menos duas versões observadoras, uma é componente e a outra é composta.

Para limitar o escopo da propagação, são definidos dois modos de propagação:

- ***restricted:*** restringe o escopo, somente as versões mais diretamente ligadas à versão modificada são atingidas pela propagação;
- ***extended:*** amplia o escopo, todas as versões ligadas direta e indiretamente à versão modificada são atingidas pela propagação.

Por *default*, a cláusula *mode* é sempre *restricted*.

Caso exista mais de uma versão do mesmo objeto versionado fazendo referência para a versão modificada, somente a versão corrente é derivada, resolvendo assim o problema da ambigüidade. Isso vale a partir do terceiro nível de composição, considerando que o primeiro nível é a versão modificada e o segundo, a versão mais diretamente ligada a versão modificada.

O modelo de versões proposto em [GOL 95], que é a base para este trabalho, não permite a exclusão de uma versão que é referenciada por outra versão. Sendo assim, no mecanismo proposto a opção *backward* não é disponível para o evento *delete*, figura 4.3.

Evento	Sentido
derive	*backward, forward, mixed*
delete	*forward*

FIGURA 4.3 − Sentido da Propagação

4.2 Modelo de Classes

Adicionalmente ao modelo de versões, originalmente proposto em [GOL 95], foram definidas cinco novas classes, ilustradas na figura 4.4 e denominadas:

- *User;*
- *NotificationSubscription;*
- *PropagationSubscription;*
- *Notification;*
- *NotificationPropagation.*

O modelo de classes descrito neste trabalho foi modelado utilizando a metodologia UML [FUR 98].

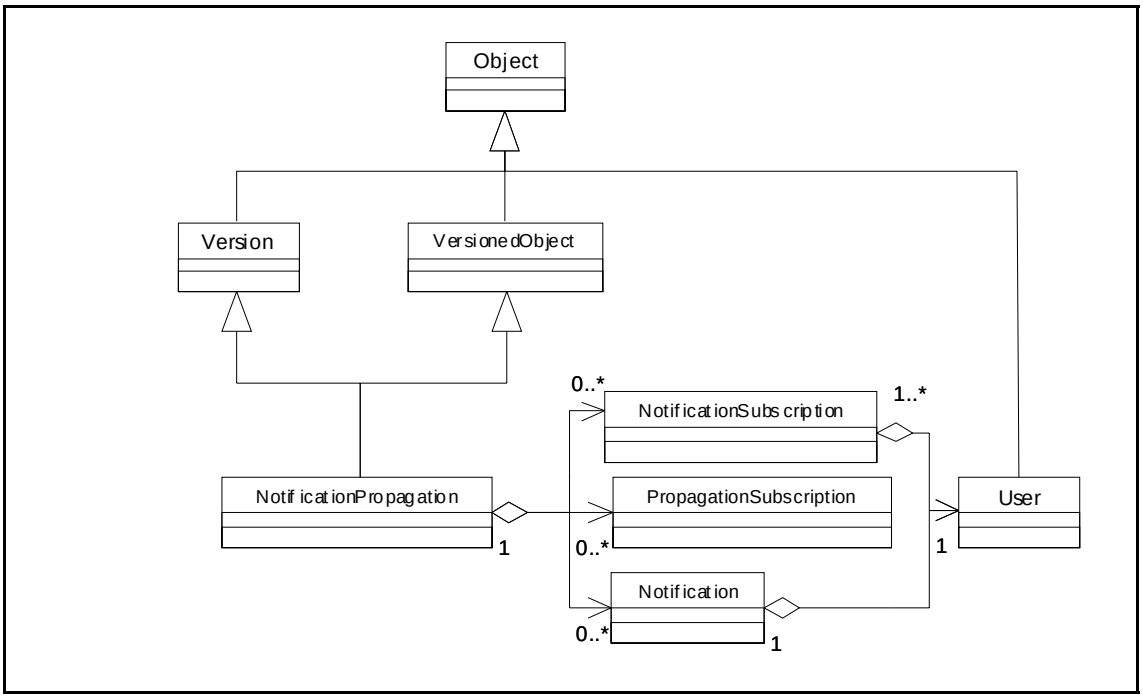

FIGURA 4.4 − Modelo de Classes

4.2.1 Classe *User*

A classe *User* é necessária para identificar o usuário no mecanismo de propagação e notificação. O "usuário" é a representação dos agentes humanos que interagem com o SGBD e podem subscrever objetos, alterar o conteúdo do banco de dados, receber e consultar notificações.

A figura 4.5 mostra a classe *User*, composta pelas propriedades:

- ***user_so:*** identificador do usuário definido no sistema operacional (utilizado para enviar mensagens na estratégia notificação ativa);
- ***user_db***: identificador do usuário no banco de dados, definido pelo SGBD, (utilizado para identificar quem executou os eventos);
- ***e−mail:*** endereço do usuário no correio eletrônico (utilizado para enviar mensagens na notificação ativa).

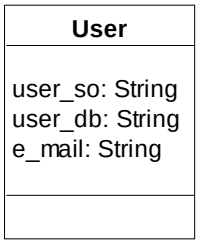

FIGURA 4.5 − Classe *User* e suas Propriedades

A fim de permitir ao usuário a utilização do mecanismo de notificação e propagação de mudanças na sua íntegra, todas as propriedades da classe *User* são de preechimento obrigatório.

As operações de manipulação de objetos da classe *User* são as operações *create_object*, *delete_object*, *modify* e *get_object* definidas no modelo de versões [GOL 95]. A operação *delete_object* não deve violar a consistência do conteúdo do banco de dados. Sendo assim, quando um usuário é excluído do banco de dados, todas as subscrições realizadas por ele também são excluídas.

4.2.2 Classe *NotificationSubscription*

Essa classe, descrita na figura 4.6, armazena as subscrições de objetos na notificação ativa e passiva para um determinado observador *(observer*). O *observer* deve ser um objeto da classe *User*.

A classe *NotificationSubscription* é composta pelas propriedades:

- ***event***: armazena o evento ocorrido;
- ***observer***: armazena os usuários que receberão a notificação;
- ***kind***: armazena o tipo de notificação: por mensagem (na tela) do sistema operacional ou por correio eletrônico.

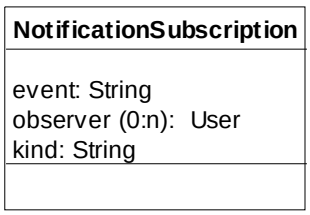

FIGURA 4.6 − Classe *NotificationSubscription* e suas Propriedades

4.2.3 Classe *PropagationSubscription*

Esta classe armazena as subscrições de objetos na propagação para um determinado observador (*observer*). O *observer* pode ser uma versão de qualquer classe do domínio da aplicação.

A figura 4.7 mostra a classe PropagationSubscription, composta pelas propriedades:

- ***event***: armazena o evento ocorrido;
- ***observer***: armazena um objeto para ser o observador;
- ***mode***: armazena o modo de propagação.

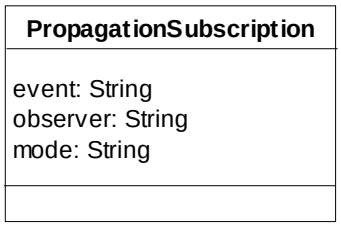

FIGURA 4.7 − Classe *PropagationSubscription* e suas Propriedades

4.2.4 Classe *Notification*

Esta classe, ilustrada na figura 4.8, armazena as notificações sobre eventos já ocorridos, subscritos para notificação passiva.

A classe *Notification* é composta pelas propriedades:

- ***event:*** armazena o evento ocorrido.
- ***timestamp:*** armazena o instante do tempo em que o objeto foi alterado;
- ***user:*** armazena o identificador do usuário que executou o evento;
- ***observer***: armazena o usuário que subscreveu para a notificação.

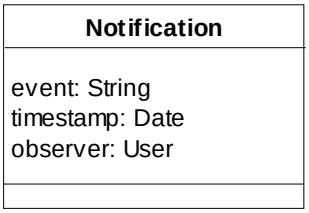

FIGURA 4.8 − Classe *Notification* e suas Propriedades

4.2.5 Classe *NotificationPropagation*

A classe *NotificationPropagation* é composta por propriedades e operações, como descrito na figura 4.9. Cada instância da classe *NotificationPropagation* contém três conjuntos de subscrições, uma para cada estratégia (*subscription_passive*, *subscription_active* e *subscription_propagation)*, e dois conjuntos de notificações sobre eventos que já ocorreram (*notificationa* e *notificationp*).

A classe NotificationPropagation é composta pelas propriedades:

- ***subscription_passive***: armazena um conjunto de subscrições para a notificação passiva;
- ***subscription_active***: armazena um conjunto de subscrições para a notificação ativa;
- ***subscription_propagation***: armazena um conjunto de subscrições para propagação;
- ***notificationp***: armazena as informações sobre as notificações (passiva) ocorridas;
- ***notificatina***: armazena as informações sobre as notificações (ativa) ocorridas.

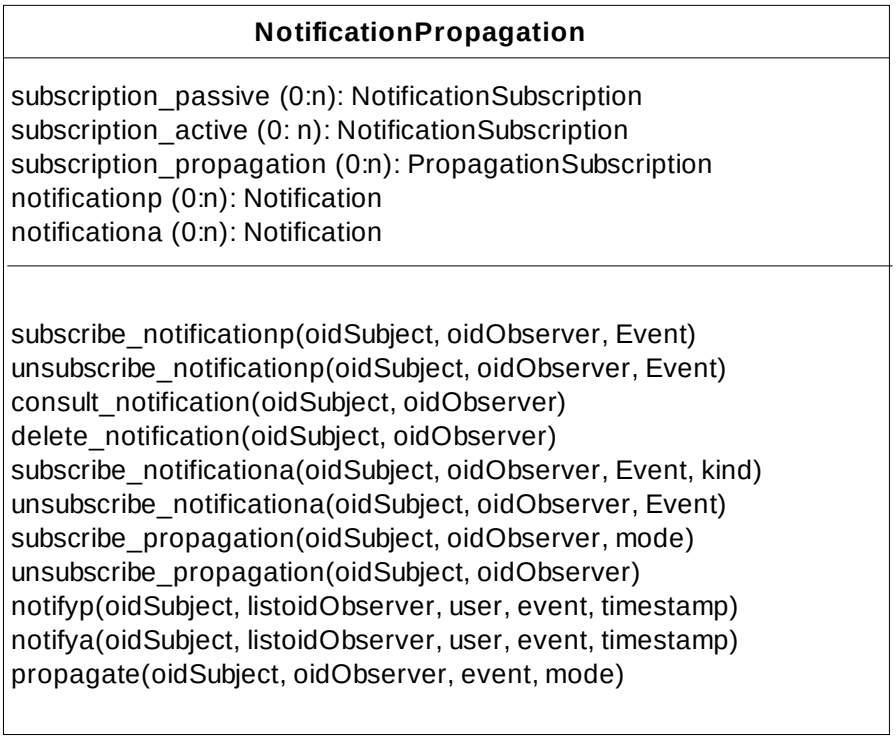

FIGURA 4.9 − Classe *NotificationPropagation* − Propriedades e Operações

4.3 Operações

Esta seção descreve as operações referentes ao mecanismo de notificação e propagação. Essas operações são incorporadas ao modelo de versões de Golendziner e estão especificadas na classe *NotificationPropagation*, figura 4.9.

Na descrição das operações propostas, foram adotadas as seguintes convenções:

- símbolos da linguagem aparecem em negrito;
- meta−símbolos aparecem normalmente, sem nenhum destaque;
- parâmetros aparecem entre parênteses;
- elementos de um conjunto aparecem entre aspas (“ ”).

Os termos *subject* (objeto observado) e *observer* (objeto observador) são utilizados para definir o papel que um objeto assume em um determinado momento durante o funcionamento do mecanismo proposto. O *subject* é o objeto que é alterado e o *observer* é o objeto que precisa ser notificado e/ou modificado. Uma mudança nos objetos observados resulta na execução de determinadas ações, como por exemplo, o envio de uma mensagem.

As operações referentes ao mecanismo são detalhadas a seguir. Cada operação é ilustrada através dos itens: sintaxe, descrição, exemplo e resultado.

4.3.1 Subscrever Objetos para Notificação Passiva

Sintaxe:

subscribe_notificationp(oidSubject, oidObserver, Event)

Descrição:

Esta operação permite subscrever um objeto na lista de notificação passiva. O parâmetro *oidSubject* contém o identificador do objeto, versão ou objeto versionado; o *oidObserver* contém o identificador do usuário que subscreve o objeto; o *Event* pode conter os seguintes eventos:

- *modify*;
- *modify*(nome do atributo);
- *change*;
- *promote*;
- *derive;*
- *delete*.

Exemplo:

subscribe_notificationp(1_motor_ 3, fonseca, modify)

Resultado:

A execução da operação resulta na inserção da versão “1_motor_3” na lista de notificação passiva, pelo usuário fonseca para o evento *modify*.

4.3.2 Cancelar Subscrição de Objetos de Notificação Passiva

Sintaxe:

unsubscribe_notificationp(oidSubject, oidObserver, Event)

Descrição:

Esta operação permite cancelar a subscrição de um objeto na lista de notificação passiva. O parâmetro *oidSubject* contém o identificador do objeto, versão ou objeto versionado; o *oidObserver* contém o identificador do usuário que cancela a subscrição do objeto; o *Event* pode conter os mesmos eventos que a operação *subscribe_notificationp*.

Esta operação não exclui as notificações já ocorridas.

Exemplo:

unsubscribe_notificationp(1_motor_ 3, fonseca, modify)

Resultado:

A execução da operação resulta na exclusão da versão “1_motor_3” da lista de notificação passiva pelo usuário “fonseca”, para o evento *modify*.

4.3.3 Consultar Mudanças

Sintaxe:

consult_notification(oidSubject, oidObserver)

Descrição:

Esta operação permite consultar os eventos realizados no objeto, versão ou objeto versionado, o usuário que realizou a mudança e o instante do tempo no qual a mudança ocorreu. O parâmetro *oidSubject* contém o identificador do objeto, versão ou objeto versionado e o *oidObserver* contém o identificador do usuário que consulta os dados.

Exemplo:

consult_notification(1_motor_ 3, fonseca)

Resultado:

A execução da operação resulta na exibição na tela dos eventos, usuários e o instante do tempo que os eventos ocorreram, conforme mostra a figura 4.10.

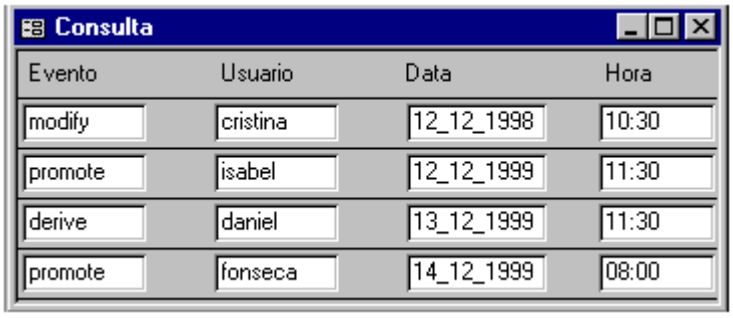

FIGURA 4.10 − Resultado da Operação *consult_notify*

4.3.4 Excluir Notificações

Sintaxe:

delete_notification(oidSubject, oidObserver)

Descrição:

Esta operação permite excluir as notificações referentes a ações já ocorridas, armazenadas na propriedade *notificationp*. O parâmetro *oidSubject* contém o

identificador do objeto, versão ou objeto versionado; o *oidObserver* contém o identificador do usuário responsável pela exclusão das notificações.

Exemplo:

delete_notification(1_motor_ 3, fonseca)

Resultado:

A execução da operação resulta na exclusão do conjunto de notificações associados à versão “1_motor_3” pelo usuário “fonseca”.

4.3.5 Subscrever Objetos para Notificação Ativa

Sintaxe:

subscribe_notificationa(oidSubject , oidObserver, Event, kind)

Descrição:

Esta operação permite subscrever um objeto na lista de notificação ativa. O parâmetro *oidSubject* contém o identificador do objeto, versão ou objeto versionado; o *oidObserver* contém o identificador do usuário que subscreve o objeto; o *Event* pode conter os mesmos eventos que a operação *subscribe_notificationp.* A informações contidas na mensagem enviada são: objeto modificado, evento correspondente, o instante em que o evento ocorreu e o usuário que executou o evento. O parâmetro *kind* especifica a forma de enviar a mensagem e pode conter as seguintes opções:

- *message*: através de uma mensagem do sistema operacional.
- *e_mail*: através de correio eletrônico;
- *all*: através das duas formas acima.

Os eventos que podem ser subscritos são os mesmos permitidos pela operação *subscribe_notificationp*.

Exemplo:

subscribe_notificationa(1_veículo_ 0, fonseca, modify, message)

Resultado:

A execução da operação resulta na subscrição do objeto “1_veículo_0” na lista de notificação ativa pelo usuário “fonseca”, para o evento *modify.*

4.3.6 Cancelar Subscrição de Objetos de Notificação Ativa

Sintaxe:

unsubscribe_notificationa(oidSubject, oidObserver, Event)

Descrição:

Esta operação pemite cancelar a subscrição de um objeto na lista de notificação ativa. O parâmetro *oidSubject* contém o identificador do objeto, versão ou objeto versionado; o *oidObserver* contém o identificador do usuário que cancela a subscrição do objeto; o *Event* pode conter os mesmos eventos que a operação *subscribe_notificationp*.

Exemplo:

unsubscribe_notificationa(1_veículo_0, fonseca, modify, delete)

Resultado:

A execução da operação resulta na exclusão da versão "1_motor_3*"* na lista de notificação ativa pelo usuário "fonseca" para o evento *modify*.

4.3.7 Subscrever Objetos para Propagação

Sintaxe:

subscribe_propagation(oidSubject, oidObserver, Event, mode)

Descrição:

Esta operação permite subscrever um objeto na lista de propagação. O parâmetro *oidSubject* contém o identificador da versão. O parâmetro *oidObserver* contém a versão que é atingida pelo evento ocorrido na versão observada. Esta versão pode ser componente ou composta da versão alterada. O parâmetro *Event* pode conter os seguintes eventos:

- *derive*;
- *delete*.

O parâmetro *mode* contém as opções para definir o escopo da propagação. São elas: *restricted* e *extended*.

Não é permitido subscrever:

- o mesmo objeto, com mesmo observador e mesmo evento;
- objetos que não possuem um relacionamento de composição.

Exemplo:

subscribe_propagation(1_motor_ 3, 1_veículo_1, derive, extended)

Resultado:

A execução da operação resulta na inserção da versão “1_motor_3” na lista de propagação para o evento *derive* com o objeto observador “1_veículo_1”. Como a opção escolhida é “*extended”* os objetos, versões ligados direta ou indiretamente à versão modificada, são afetados pelas mudanças.

4.3.8 Cancelar Subscrição de Objetos de Propagação

Sintaxe:

unsubscribe_propagation(oidSubject, oidOberver, Event)

Descrição:

Esta operação permite cancelar a subscrição de um objeto na lista de propagação. O parâmetro *oidSubject* contém o identificador da versão; *oidObserver* contém o identificador da versão que é alterada pelo mecanismo após a ocorrência de um determinado evento; *Event* pode conter os mesmos eventos que a operação *subscribe_propagation*.

Exemplo:

unsubscribe_propagation(1_motor_ 3, 1_veículo_1, derive, fonseca)

Resultado:

A execução da operação resulta na exclusão da versão observada “1_motor_3” na lista de propagação para o evento *derive* e para a versão observadora “1_veículo_1”.

4.3.9 Incluir Ocorrência de Notificações

Sintaxe:

notifyp(oidSubject, listoidObserver, user, event, timestamp)

Descrição:

Esta operação permite inserir as informações sobre os eventos que ocorreram na lista de notificação passiva. A lista é representada pela propriedade *notificationp* pertencente à classe *NotificationPropagation*. O parâmetro *oidSubject* contém o identificador do objeto, versão ou objeto versionado; o *oidObserver* contém o identificador do usuário observador; o *user* contém o identificador do usuário que causou a mudança; o *event* contém o tipo de evento ocorrido e *timestamp* contém a hora e a data em que o evento ocorreu.

Exemplo:

notifyp(1_motor_3, “isabel, cristina”, fonseca, derive, “12/12/ 1998, 10:30”)

Resultado:

A execução da operação resulta na inserção de uma nova notificação no conjunto de notificação passiva com as informações da versão modificada:

"1_motor_3", usuários observadores: isabel, cristina, evento: *derive* e o *timestamp*: "12/12/ 1998, 10:30".

4.3.10 Notificar Usuários

Sintaxe:

notifya(oidSubject, listoidObserver, user, event, timestamp, kind)

Descrição:

Esta operação envia uma mensagem para os usuários que subscreveram o objeto modificado. O parâmetro *oidSubject* contém o identificador do objeto, versão ou objeto versionado; o *listoidObserver* contém o conjunto de identificadores dos usuários observadores; o *user* contém o identificador do usuário que modificou o objeto; o *event* contém o tipo de evento ocorrido. As informações contidas na mensagem enviada são: objeto modificado, evento correspondente, o instante em que o evento ocorreu e o usuário que executou o evento. O parâmetro *kind* especifica a forma de enviar a mensagem e pode conter as seguintes opções:

- *message*: através de uma mensagem do sistema operacional, como mostra a figura 4.11;
- *e_mail*: através de correio eletrônico;
- *all*: através das duas formas acima.

Exemplo:

notifya(1_motor_1, "fonseca, isabel", cristina, modify, "12/12/ 1998, 10:30", message)

Resultado:

A execução da operação resulta no envio de uma mensagem para os usuários fonseca e isabel, com as informações sobre as mudanças na versão "1_motor_1", conforme ilustra a figura 4.11.

FIGURA 4.11 − Exemplo de Notificação Ativa

4.3.11 Propagar Mudanças

Sintaxe:

propagate(oidSubject, oidObserver, event, mode)

Descrição:

Esta operação reflete as mudanças no objeto observador, ocorridas no objeto observado. O parâmetro *oidSubject* contém o identificador da versão observada; o *oidObserver* contém o objeto do qual é derivada uma nova versão ou que é excluído; o *event* contém o evento ocorrido e “mode” pode conter as opções *restricted* ou *extended*.

Exemplo:

propagate(1_motor_2, 1_veículo_1, derive, restricted)

Resultado:

A execução da operação resulta na criação de uma nova versão do objeto “1_veículo_1” com referência à nova versão “1_motor_2”.

4.4 Um Exemplo

A figura 4.12 mostra o diagrama de classes usado nos exemplos apresentados neste trabalho. O diagrama ilustra as classes *Veiculo*, *Motor* e *Valvula*, as quais possuem um relacionamento de composição.

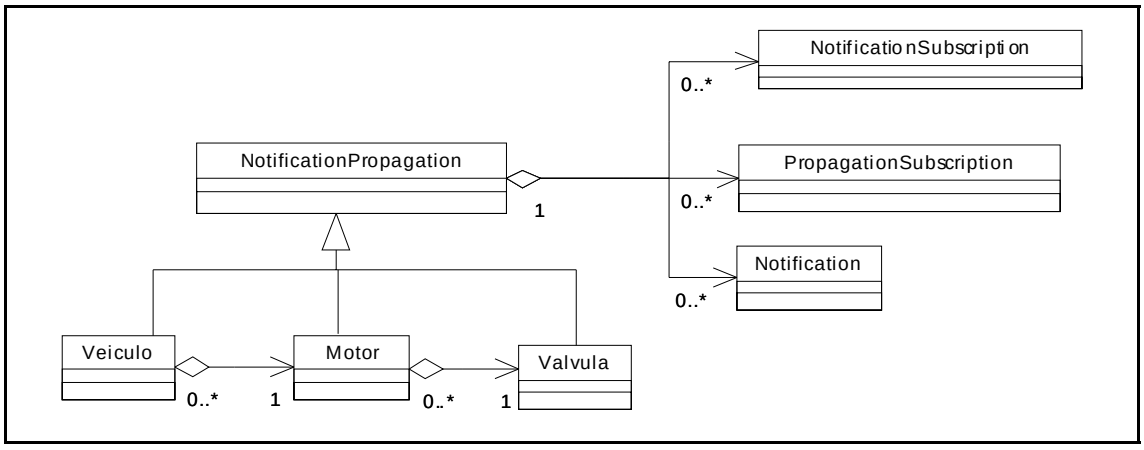

FIGURA 4.12 − Projeto de Veículos

Para ilustrar o uso do mecanismo proposto, toma−se como exemplo um ambiente de projeto de veículos e seus componentes. Em um ambiente como este, diversos usuários trabalham em grupo visando a alcançar um objetivo comum. Sendo assim, é necessário possibilitar a divulgação das ações tomadas por esses usuários e a propagação das mudanças ocorridas.

4.4.1 Notificação Passiva

Esta seção apresenta um exemplo que ilustra o uso do mecanismo proposto, utilizando a estratégia notificação passiva.

Considera−se que os engenheiros Cristina e Daniel trabalham no projeto de veículos, enquanto Isabel trabalha no projeto de motores. Quando um novo veículo é projetado, o engenheiro responsável pelo projeto de veículos deve conhecer os tipos de motores existentes. Portanto, toda vez que uma nova versão de um determinado motor é criada, é desejável que o SGBD mantenha persistentes os dados referentes à ação realizada.

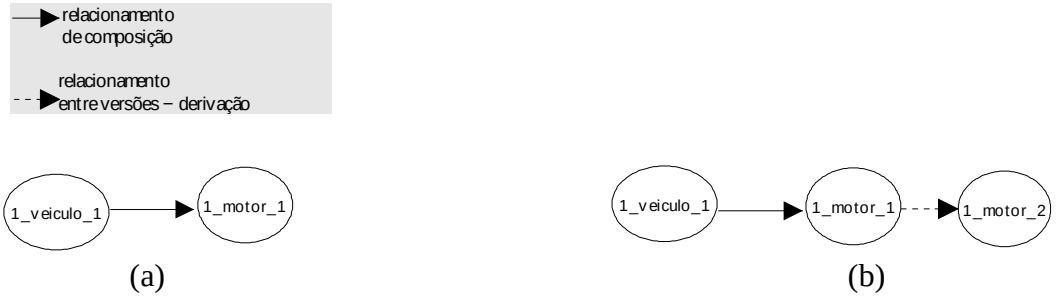

FIGURA 4.13 − Criação de Versão com Notificação

As etapas apresentadas a seguir descrevem o funcionamento do mecanismo de notificação passiva considerando o ambiente descrito acima.

1. Supõe−se que Cristina tem necessidade de acompanhar a evolução de um determinado motor. Cristina subscreve o 1_motor_1 na lista de notificação passiva, utilizando a seguinte operação: **subscribe_notificationp** (1_motor_ 1, cristina, modify). Essa operação é responsável pela inclusão das informações passadas como parâmetros no conjunto de notificação passiva. A propriedade *subscription_passive*, pertencente à classe *NotificationPropagation,* guarda as informações referentes à subscrição.

2. Isabel deriva uma nova versão do 1_motor_1, conforme ilustrado na figura 4.13(b), criando a versão 1_motor_2 através da operação **derive_version** (1_motor_1). Um procedimento que verifica se o evento **derive** está subscrito para o motor 1.1 é excecutado. Caso verdadeiro, as informações: usuário, evento, objeto e *timestamp* são inseridas na lista *notificationp*. O método utilizado para isto é **notifyp**(1_motor_1, "cristina", isabel, derive, "12/12/ 1998, 10:30").

    1. Quando uma nova versão de automóvel for projetada, o engenheiro interessado, neste caso Cristina, pode consultar os dados referentes ao motor usando a operação **consult_notify**(1_motor_1, cristina). Essa operação mostra o usuário responsável pela mudança, o momento em que ocorreu a mudança, o evento e a versão modificada, conforme ilustrado na figura 4.14.

    2. Se o usuário Cristina desejar cancelar as subscrições, deve ser utilizada a operação **unsubscribe_notificationp**(1_motor_1, cristina, modify). Após a execução desta operação, a versão 1_motor_1 ao derivar novas versões, não é monitorada.

    3. A operação acima não exclui as notificações referentes aos eventos que já ocorreram, para isso deve ser utilizada a operação **delete_notification** (1_motor_1, fonseca).

versão	*evento*	*usuário*	*timestamp*
1_motor_1	derive	isabel	10:30 12/12/1999

FIGURA 4.14 − Consulta − Notificação Passiva

4.4.2 Notificação Ativa

Esta seção apresenta um exemplo ilustrando o uso do mecanismo com a estratégia de notificação ativa.

Os engenheiros Cristina e Daniel, por exemplo, trabalham no projeto de veículos enquanto Isabel trabalha no projeto de motores. O engenheiro responsável deve saber quando um determinado motor em desenvolvimento é desconsiderado. Diferente do exemplo da seção 4.4.1, toda vez que uma nova versão de um determinado motor for excluída, é desejável que o sistema de banco de dados envie uma mensagem para os usuários interessados.

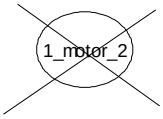

FIGURA 4.15 − Exclusão de Versão

As etapas descritas a seguir descrevem o funcionamento do mecanismo de notificação ativa:

1. Supondo que Cristina necessita ser informada quando um determinado motor é excluído, então ela deve subscrever o motor 1.2 na lista de notificação ativa utilizando a seguinte operação: **subscribe_notificationa**(1_motor_2, cristina,"delete", message). Essa operação é responsável pela inclusão das informações na lista *subscription_active*, propriedade da classe *NotificationPropagation.*

    1. Quando Isabel excluir a versão do 1_motor_2, figura 4.15, utilizando a operação **delete_object**(1_motor_2) um *trigger* deve ser disparado. Esse trigger executa um método que verifica se o evento **delete** está subscrito para o motor 1.2. Caso verdadeiro, o mecanismo exibe uma mensagem, figura 4.16, com as informações: usuário, evento, objeto e *timestamp*, utilizando o método **notifya**(1_motor_2, "cristina", isabel, delete, "12/12/ 1998, 10:30", message).

    2. Se o usuário deseja cancelar subscrições, deve utilizar a operação **unsubscribe_notificationa**(1_motor_2, cristina, delete).

FIGURA 4.16 − Exemplo de Notificação Ativa

4.4.3 Propagação

Esta seção apresenta um exemplo que ilustra o uso do mecanismo proposto, utilizando a estratégia propagação.

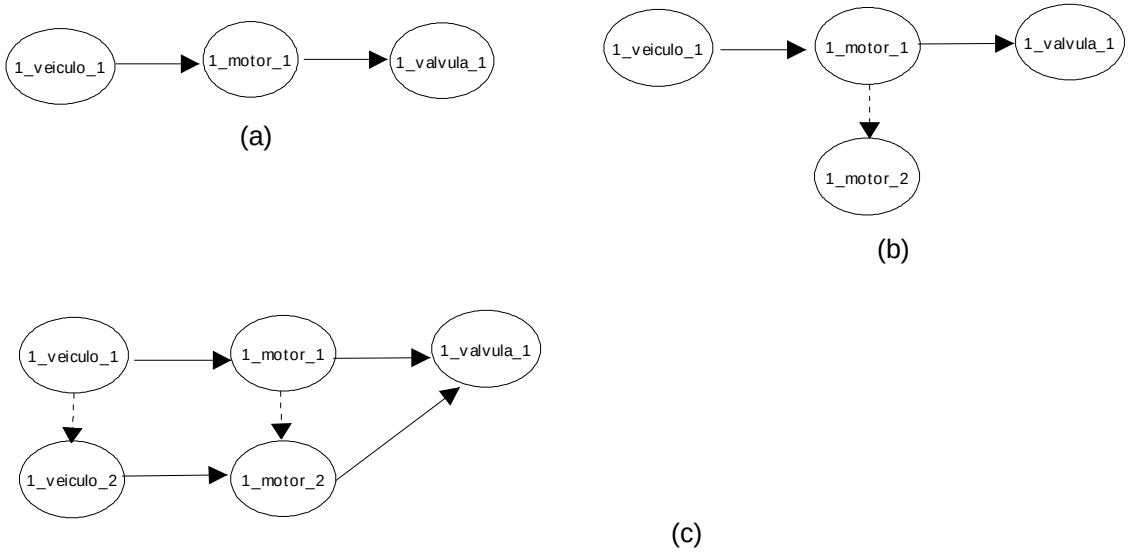

FIGURA 4.17 − Propagação de Versões − Modo *restricted*

Suponha−se que toda vez que uma nova versão do motor “1_motor_1” é criada, uma nova versão de veículo também deve ser criada automaticamente pelo sistema. As etapas descritas a seguir descrevem o funcionamento do mecanismo para esse exemplo.

1. O usuário deve subscrever o motor 1.1 na lista de propagação usando a seguinte operação: **subscribe_propagation**(1_motor_1, 1_veiculo_1, derive, restricted). Essa operação é responsável pela inclusão das informações na lista *subscription_propagation*, propriedade da classe *NotificationPropagation.*

2. Uma nova versão do motor 1.1 é derivada, conforme ilustrado na figura 4.17(b), criando a versão 1.2 através da operação **derive_version** (1_motor_1). Um procedimento que verifica se o evento **derive** está subscrito para o 1_motor_1 é executado. Caso verdadeiro, o mecanismo cria uma nova versão do objeto observador, utilizando o método **propagate**(1_motor_2, 1_veículo_1, derive, restricted). Como o modo escolhido é *restricted*, somente a versão ligada diretamente à versão modificada é criada, figura 4.17(c);

3. Se o usuário deseja cancelar subscrições, é utilizada a operação **unsubscribe_propagation**(1_motor_1, 1_veículo_1, derive);

O objeto é criado fazendo referência a mesma versão do objeto componente do qual ele foi derivado.

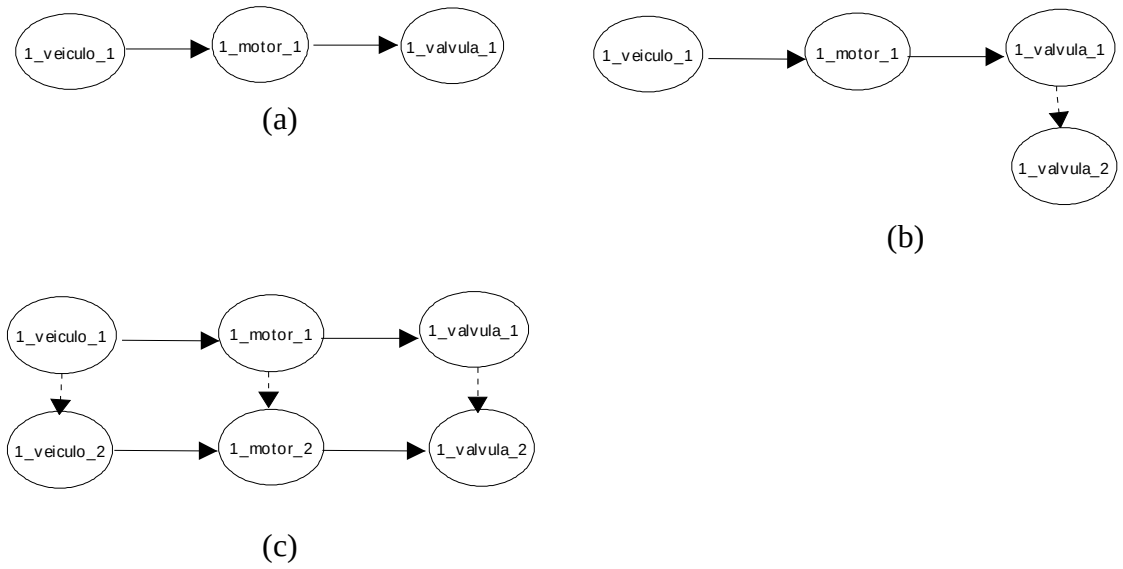

FIGURA 4.18 − Propagação de Versões − Modo *extended*

Suponha−se que toda vez que uma nova versão do objeto “1_valvula_1” é criada, uma nova versão de motor e de todos as versões ligadas diretas e indiretamente a ela devem ser criadas automaticamente pelo sistema. As etapas descritas a seguir descrevem o funcionamento do mecanismo para este exemplo:

1. O usuário deve subscrever a versão “1_valvula_1” na lista de propagação usando a seguinte operação: **subscribe_propagation**(1_valvula_1, 1_motor_1, derive, extended). Essa operação é responsável pela inclusão das informações na lista *subscription_propagation*, propriedade da classe *NotificationPropagation.*

2. Uma nova versão de “1_valvula_1” é derivada, conforme ilustrado na figura 4.18(b), criando a versão 1.2 através da operação **derive_version** (1_valvula_1). Um *trigger* é disparado, executando um método que verifica se o evento **derive** está subscrito para a versão “1_valvula_1”. Caso verdadeiro, o mecanismo cria uma nova versão da versão observadora, utilizando o método **propagate**(1_valvula_2, 1_motor_1, derive, extended). Como modo de propagação escolhido é *extended*, uma nova versão de veículo também é criada, figura 4.18(c).

4.5 Conclusões

Neste capítulo, foi especificado um mecanismo de notificação e propagação de mudanças proposto para o modelo de versões. Assim, três estratégias foram definidas, combinando as abordagens propostas por dois autores: Chou e Oussalah. Chou [CHO 88] apresenta a notificação ativa e passiva; Oussalah a técnica de propagação [OUS 96, OUS 97].

Propagação de versões pode causar proliferação de versões. Para resolver este problema as operações *subscribe* e *unsubscribe* foram utilizadas. Com isso, somente as versões subscritas são atingidas pela propagação.

Outra forma de limitar o número de versões criadas foi definir se as versões diretamente ligadas à versão modificada são atingidas pelo propagação. Isso é possível declarando a cláusula *mode* como *restricted* na subscrição de objetos para a propagação.

Oussalah define três sentidos possíveis para a abordagem propagação: *forward*, *backward* e *mixed*. Esses três sentidos foram utilizados, neste trabalho, para permitir ao usuário, especificar quais versões são atingidas pela propagação: se somente as versões componentes (*forward*), ou somente versões compostas (*backward*) ou ambas (*mixed*).

Na definição das operações, os termos observador e observado foram usados. Tais termos permitem especificar o papel que um determinado objeto assume durante o funcionamento do mecanismo. Essa nomenclatura é usada no padrão *Observer* [GAM 99].

O mecanismo sugerido como trabalho futuro em [GOL 95] visava a tratar somente a evolução de versões de objetos componentes. Analisando alguns sistemas de banco de dados comerciais, foi verificada que a existência de mecanismos abrangentes. Esses mecanismos tratam mudanças em qualquer objeto, independente se este é versão ou objeto versionado. A proposta descrita aqui utiliza esta abordagem tratando mudanças ocorridas em objetos, versões e objetos versionados.

Versões de referências entre objetos são definidas no modelo conceitual hipermídia [SOA 95] e versões de elos (referências entres documentos) são definidas no modelo de Oussalah [OUS 97]. Propagação de relacionamentos são tratadas por essas propostas. O modelo de versões de Golendziner não provê versões de relacionamentos, sendo desnecessário um mecanismo para tratar propagação deste tipo de versão.

Projetos de pesquisa [CHO 88, KAT 90, OUS 97, URT 98] não especificam a representação do agente (usuário que envia e recebe mensagens) utilizada no ambiente de banco de dados. Alguns sistemas de *workflow* utilizam um próprio gerenciador de usuários que armazena e controla seus identificadores no sistema e as suas respectivas senhas. Neste trabalho, optou−se por definir uma classe que permitisse armazenar informações acerca dos agentes. Dependendo da estratégia escolhida, uma representação é utilizada. Por exemplo, a estratégia de notificação ativa usa a representação de *e−mail* do correio eletrônico.

A tabela 4.4, apresenta uma comparação entre o mecanismo proposto e os presentes em algumas propostas descritas na seção 2.1.

As tabelas 4.5(a), 4.5(b) e 4.5(c) apresentam uma comparação entre o mecanismo proposto e os presentes em alguns SGBDs descritos na seção 2.2.

TABELA 4.4 − Comparação entre os Mecanismos Analisados e o Mecanismo Proposto

Propostas encontradas em Projetos de Pesquisa						
Características	**ATWOOD**	**CHOU E KIM**	**KATZ**	**OUSSALAH**	**SOARES**	**MECANISMO PROPOSTO**
Solução para resolver o problema do escopo da notificação ou propagação	Não apresenta	Restrições: somente as versões diretamente ligadas às versões alteradas são modificadas	Restrições: de configuraçõe s	− Atributos: sensíveis e não sensíveis − Versões: sensíveis ou não sensíveis − Restrições: somente as versões com estado permanente propagam mudanças; − Atributo: modo e sentido	− Atributos: versionável e não versionável − Restricões: somente as versões ligadas diretamente a versão modificada são propagadas − Operação: *chek_in_one*	− Atributo: modo e sentido − Restrições: versões diretamente ligadas a versão modificada são atingidas − Operações: *subscribe* e *unsubscribe*.
Solução para resolver o problema da ambigüidade	Não apresenta	*	Operações: *chek_in* e *chek−out* em grupo	Atributo: *multipath*	Não apresenta	− Operações: *subscribe* e *unsubscribe*

* não especificado na literatura consultada.

TABELA 4.5 (a) − Comparação entre os Mecanismos Analisados e o Mecanismo Proposto

SISTEMAS DE GERENCIAMENTO DE BANCO DE DADOS					
Características	**ITASCA**	**OBJECTSTORE**	**O2**	**GEMSTONE**	**MECANISMO PROPOSTO**
Tipos de estatrégias	− notificação passiva − notificação ativa	notificação ativa	notificação ativa	notificação ativa (mecanismo de notificação e mecanismo sinalizador de troca)	− notificação passiva − notificação ativa − propagação
Trata as mudanças ocorridas	− em objetos − em versões	− em objetos − em segmentos	− em objetos − em versões	em objetos	− em versões − em objetos − em objetos versionados
Eventos que acionam o mecanismo	− alteração de objetos − exclusão de objetos − criação de objetos − *check−in* e *check−out* de objetos − alteração de versões − exclusão de versões − criação de versões − *check−in* e *check−out* de versões	− alteração de objetos − exclusão de objetos − inicialização de uma sessão do banco de dados − finalização de uma sessão de banco de dados	− alteração de objetos − exclusão de objetos − alteração de versões − exclusão de versões − inicialização de uma sessão do banco de dados − finalização de uma sessão de banco de dados − ocorrência de um evento definindo pelo usuário		− alteração de objetos − exclusão de objetos − alteração de objetos − exclusão de versões − criação de versões − promoção de versões − alteração de versão corrente

* não especificado na literatura consultada.

TABELA 4.5 (b) − Comparação entre os Mecanismos Analisados e o Mecanismo Proposto − Continuação

SISTEMAS DE GERENCIAMENTO DE BANCO DE DADOS					
Características	**ITASCA**	**OBJECTSTORE**	**O2**	**GEMSTONE**	**MECANISMO PROPOSTO**
Ações executadas pelo mecanismo como resposta à eventos	− uma mensagem é enviada ao usuário ou uma ação definida pelo usuário é executada (notificação ativa) − uma lista de mudanças é mantida pelo sistema (notificação passiva)	implementadas na aplicação	implementadas na aplicação	implementadas na aplicação	− uma mensagem é enviada ao usuário (notificação ativa) − uma lista de mudanças é mantida pelo sistema (notificação passiva) − mudanças nos dados do banco de dados (propagação)
Acrescenta atributo a classe notificável	acrescenta *timestamp*	*	*	*	acrescenta *timestamp*
Implementação do mecanismo	funções definidas em LISP	métodos definidos nas API C++ e API Java	− módulo *O2 Notification* − não especifica a linguagem, se é *: C++, java ou O2C*	GCI (GEMSTONE C Interface)	− PL/SQL − triggers

* não especificado na literatura consultada.

TABELA 4.5 (c) − Comparação entre os Mecanismos Analisados e o Mecanismo Proposto − Continuação

SISTEMAS DE GERENCIAMENTO DE BANCO DE DADOS					
Características	**ITASCA**	**OBJECTSTORE**	**O2**	**GEMSTONE**	**MECANISMO PROPOSTO**
Armazenamento das Notificações	− as notificações são persistentes até que o usuário "aprove" a mudança (notificação passiva) − não é persistente (notificação ativa)	não são persistentes	são armazenadas no servidor	não são persistentes	− são armazenadas (persistente) em uma lista ligada ao objeto (notificação passiva) − não é persistente (notificação ativa)
Armazenamento das subscrições	*	não são persistentes	são armazenadas no servidor e não são persistentes	*	são persistentes para todas as estratégias

* não especificado na literatura consultada.

5 Implementação

Neste capítulo, é apresentada a implementação do mecanismo de notificação e propagação, proposto no capítulo 4. Esta implementação é definida para o sistema Oracle 8. A versão utilizada foi 8.0.4 (plataforma Windows NT). As operações e *triggers* foram implementados na linguagem PL/SQL.

Para facilitar a identificação de tipos, tabelas e *tablespaces* na criação e utilização destes, a seguinte nomenclatura é utilizada:

- nome de tipo *VA*RRAY é seguido de traço (_) e a palavra VA, por exemplo, Event_VA;
- nome de pacote (*PA*CKAGE) é seguido de traço (_) e a palavra PA, por exemplo NotificationPassive_PA;
- nome de *tablespace* é seguido de traço (_) e a palavra TS, por exemplo Mechanism_TS.

5.1 Tabelas

As classes do mecanismo de notificação e propagação são mapeadas para tabelas no banco de dados Oracle. Nesta seção são apresentadas as tabelas utilizadas na implementação do mecanismo proposto.

5.1.1 Tabela *Users*

A classe *User* do modelo de classes, definida na seção 4.2.1, é mapeada para uma tabela no banco de dados Oracle. As propriedades desta classe, user_so, user_db, e_mail, são mapeadas para colunas descritas na tabela 5.1. Uma coluna adicional é criada, oid_user, para representar o identificador de objeto definido no modelo de versões.

TABELA 5.1 − Tabela *Users*

Coluna	**Tipo de Dados**
oid_user	VARCHAR2(30)
user_so	VARCHAR2(80)
user_db	VARCHAR2(80)
e_mail	VARCHAR2(80)

A palavra *User* é uma palavra reservada da linguagem PL/SQL [GOK 98]. Por este motivo, na implementação do mecanismo proposto, a tabela que mapeia a classe *User* do modelo de classes passa a ser denominada *Users* e o atributo que mapeia a propriedade *user*, da tabela *Notification*, passa a ser *users*.

A figura 5.1 contém o *script* para criação da tabela *Users* e o *trigger* deleteUser. Este *trigger* exclui todas as subscrições realizadas pelo usuário, quando este é excluído.

```
CREATE TABLE Users (
        oid_user VARCHAR2(30) PRIMARY KEY,
        users_so VARCHAR2(80) NOT NULL,
        user_db VARCHAR2(80) NOT NULL,
        e_mail VARCHAR2(80) NOT NULL);

CREATE OR REPLACE TRIGGER deleteUser
        AFTER DELETE ON Users FOR EACH ROW
BEGIN
    DELETE FROM NotificationSubscription
          WHERE oid_observer = oid_user
END deleteUser;
```

FIGURA 5.1 − Tabela *Users*

5.1.2 Tabela *NotificationSubscription*

A classe *NotificationSubscription* do modelo de classes, especificada na seção 4.2.2, é mapeada para uma tabela no banco de dados Oracle. As propriedades desta classe, *observer, kind, event* são mapeadas para colunas, conforme mostra a tabela 5.2.

São criadas três colunas adicionais:

- oid_subject − representa o identificador de objeto do modelo de versões;
- attr_modify − permite a subscrição de atributos para o evento *modify*.
- kind − armazena o tipo de notificação, que pode ser passiva ou ativa.

TABELA 5.2 − Tabela *NotificationSubscription*

Coluna	**Tipo de Dados**
oid_subject	VARCHAR2(30)
oid_observer	VARCHAR2(30)
kind	VARCHAR2(1)
event	VARCHAR2(30)
attr_modify	VARCHAR2(30)

A figura 5.2 contém o *script* para criação da tabela *NotificationSubscription*.

```
CREATE TABLE NotificationSubscription (
        oid_subject VARCHAR2(30) NOT NULL,
        oid_observer VARCHAR2(30) NOT NULL,
        kind VARCHAR2(1) NOT NULL,
        event VARCHAR2(30) NOT NULL,
        attr_modify VARCHAR2(30),
        PRIMARY KEY (oid_subject, oid_observer, kind, event),
        FOREIGN KEY (oid_observer) REFERENCES Users(oid_user));
```

FIGURA 5.2 − Tabela *NotificationSubscription*

5.1.3 Tabela *PropagationSubscription*

A classe *PropagationSubscription* do modelo de classes, definida na seção 4.2.3, é mapeada para uma tabela no banco de dados Oracle. As propriedades desta classe, *observer, event, mode* são mapeadas para colunas, conforme mostra a tabela 5.3. Uma coluna adicional é criada, oid_subject, para representar o identificador de objeto do modelo de versões.

TABELA 5.3 − Tabela *PropagationSubscription*

Coluna	**Tipo de Dados**
oid_subject	VARCHAR2(30)
oid_observer	VARCHAR2(30)
event	VARCHAR2(30)
modo	VARCHAR2(10)

A palavra *mode* é uma palavra reservada da linguagem PL/SQL. Por este motivo, na implementação do mecanismo proposto, o atributo que mapeia a propriedade *mode* do modelo de classes passa a ser denominada *modo*.

A figura 5.3 contém o script para criação da tabela *PropagationSubscription*.

```
CREATE TABLE PropagationSubscription(
        oid_subject VARCHAR2(30) NOT NULL,
        oid_observer VARCHAR2(30) NOT NULL,
        event VARCHAR2(30) NOT NULL,
        modo VARCHAR2(10),
        PRIMARY KEY (oid_subject, oid_observer, event, modo));
```

FIGURA 5.3 − Tabela *PropagationSubscription*

5.1.4 Tabela *Notification*

A classe *Notification* do modelo de classes, definida na seção 4.2.4, é mapeada

para uma tabela no banco de dados Oracle. As propriedades desta classe, *user*, *observer*, *event*, *timestamp*, são mapeadas para colunas, conforme mostra a tabela 5.4. Uma coluna adicional é criada, oid_subject, para representar o identificador de objeto do modelo de versões. A coluna attr_modify é criada para permitir a subscrição de atributos para o evento *modify*. A coluna kind é criada para armazenar o tipo de notificação.

TABELA 5.4 − Tabela *Notification*

Coluna	**Tipo de Dados**
oid_subject	VARCHAR2(30)
oid_user	VARCHAR2(30)
oid_observer	VARCHAR2(30)
event	VARCHAR2(30)
atrr_modify	VARCHAR2(30)
Kind	VARCHAR2(1)
timestamp	DATE

A figura 5.4 contém o *script* para criação da tabela *Notification*.

```
CREATE TABLE Notification (
        oid_subject VARCHAR2(30) NOT NULL,
        oid_user VARCHAR2(30) NOT NULL,
        oid_observer VARCHAR2(30) NOT NULL,
        event VARCHAR2(30) NOT NULL,
        attr_modify VARCHAR2(30),
        kind VARCHAR2(1) NOT NULL,
        timestamp DATE NOT NULL,
        PRIMARY KEY (oid_subject, oid_user, oid_observer, event, timestamp),
        FOREIGN KEY (oid_user) REFERENCES Users(oid_user),
        FOREIGN KEY (oid_observer) REFERENCES Users(oid_user));
```

FIGURA 5.4 − Tabela *Notification*

5.2 Operações do mecanismo

Todas as operações apresentadas para o mecanismo de notificação passiva foram implementadas como procedimentos, em PL/SQL, no pacote NotificationPassive_PAC (anexo 3).

No sistema Oracle, pacote é um grupo de funções e procedimentos relacionados entre si, armazenados juntamente no banco de dados para serem utilizados como uma unidade de código. As funções e os procedimentos contidos em um pacote podem ser

chamados explicitamente por aplicações ou usuários [GOK 98]. Um pacote é criado em duas partes: a especificação e o corpo. Todas as construções públicas são declaradas na especificação do pacote e todas as construções públicas e privadas são definidas no corpo do pacote.

5.2.1 subscribe_notificationp(oidSubject, oidObserver, Event) e unsubscribe_notificationa(oidSubject , oidObserver, Event)

Estas operações foram propostas como procedimentos que permitem a subscrição de objetos e seu cancelamento, para notificação passiva. Os valores passados como parâmetros são inseridos (*subscribe*) e excluídos (*unsubscribe*) da tabela NotificationSubscription, assim como o tipo de notificação (variável local *kind*).

O código das operações referentes à notificação passiva também podem ser utilizados para implementar a notificação ativa, sendo necessário adicionar um parâmetro que informe o tipo de notificação, que pode ser por mail ou por mensagens na tela.

A figura 5.5 mostra o código dos procedimentos subscribe_notificationp e unsubscribe_notificationp.

```
PROCEDURE subscribe_notificationp(oidSubject IN VARCHAR2,
                                  oidObserver IN VARCHAR2,
                                  ev IN Event_VA)
IS
BEGIN
        IF ev.COUNT > 1 THEN
                INSERT INTO NotificationSubscription (oid_subject, oid_observer, kind, event,
attr_modify)
                        VALUES (oidSubject, oidObserver,'p', ev(1), ev(2));
        ELSE
                INSERT INTO NotificationSubscription (oid_subject, oid_observer, kind, event)
                        VALUES (oidSubject, oidObserver,'p', ev(1));
        END IF;
END;

PROCEDURE unsubscribe_notificationp(oidSubject IN VARCHAR2,
                                    oidObserver IN VARCHAR2,
                                    ev IN Event_VA)
IS
BEGIN
 DELETE FROM NotificationSubscription
        WHERE oid_subject = oidSubject AND
            oid_observer = oidObserver AND
            kind = 'p' AND
            event = ev(1);
END;
```

FIGURA 5.5 − Procedimentos *subscribe_notificationp* e *unsubscribe_notificationp*

5.2.2 notifyp(oidSubject, listoidObserver, user, event, timestamp)

Esta operação foi proposta como o procedimento notifyp_all_observers (figura 5.6) que permite armazenar, na tabela Notification, as informações sobre os eventos

ocorridos nos dados do banco.

```
PROCEDURE notifyp_all_observers(oidSubj IN VARCHAR2,
                                ev IN Event_VA,
                                timestamp IN DATE)
 IS
 CURSOR all_observers_cur
        IS
        SELECT oid_observer
        FROM   NotificationSubscription
        WHERE  oid_subject = oidSubj AND
           kind = 'p' AND
           event = ev(1);
oid_user_var VARCHAR2(30);
BEGIN
        SELECT oid_user INTO oid_user_var FROM Users WHERE user_db = USER;
        FOR all_observers_var IN all_observers_cur LOOP
                NotificationPassive_PAC.notifyp(oidSubj,all_observers_var.oid_observer,
                                                oid_user_var, ev, timestamp);
        END LOOP;
END;
```

FIGURA 5.6 − Procedimento *notifyp_all_observers*

O *trigger* "deletemotor" (anexo 5) contém uma chamada ao procedimento notifyp_all_observers. Este *trigger* gera uma notificação sempre que uma exclusão de um objeto subscrito ocorre. As operações (anexo 4) *promote, derive, change_current* também possuem uma chamada ao procedimento notifyp_all_observers.

O processo de notificação do evento *modify* difere dos demais, por requerer a informação de qual atributo foi alterado. Para este caso, o procedimento auxiliar *notifyp* (figura 5.7) foi implementado.

```
PROCEDURE notifyp(oidSubj IN VARCHAR2,
                  oidObserv IN VARCHAR2,
                  oidUser IN VARCHAR2,
                  ev IN Event_VA,
                   timestamp IN DATE)
 IS
BEGIN
        IF ev.COUNT > 1 THEN
                INSERT INTO Notification
                (oid_subject, oid_user, oid_observer, event, attr_modify, kind, data)
                VALUES
          (oidSubj, oidUser,  oidObserv, ev(1),  ev(2), 'p',  timestamp);
        ELSE
                INSERT INTO Notification
                (oid_subject, oid_user, oid_observer, event, kind, data)
                VALUES
          (oidSubj, oidUser,  oidObserv,    ev(1), 'p',  timestamp);
        END IF;
END;
```

FIGURA 5.7 − Procedimento *notifyp*

5.2.3 consult_notification(oidSubject, oidObserver)

Esta operação foi proposta como o procedimento consult_notification e permite consultar as notificações sobre os eventos ocorridos nos dados do banco (figura 5.8).

```
PROCEDURE consult_notification(oidSubject IN VARCHAR2,
                               oidObserver IN VARCHAR2)
IS
CURSOR all _notifications_cur RETURN Notification%ROWTYPE
 IS
   SELECT * FROM Notification
   WHERE oid_subject = oidSubject AND
         oid_observer = oidObserver;
BEGIN
      SET SERVEROUTPUT ON;
      FOR all_notifications_var IN all_notifications_cur LOOP
            DBMS_OUTPUT.PUT('User:'  || all_notifications_var.OID_USER);
            DBMS_OUTPUT.PUT('  Date:' || TO_CHAR(all_notifications_var.DATA,
                                          'DY DD−MON−YYYY HH24:MI:SS'));
            DBMS_OUTPUT.PUT('  Event:' || all_notifications_var.event);
            IF (all_notifications_var.event = 'modify')
            THEN
                  DBMS_OUTPUT.PUT_LINE('Atributo:'||
                                          all_notifications_var.attr_modify);
            ELSE
                  DBMS_OUTPUT.PUT_LINE('');
            END IF;
      END LOOP;
END;
```

FIGURA 5.8 − Procedimento *consult_notification*

5.2.4 delete_notification(oidSubj, oidObserver)

Esta operação foi proposta como o procedimento delete_notification e permite excluir as notificações armazenadas na Tabela Notification (figura 5.9).

```
PROCEDURE delete_notification(oidSubject IN VARCHAR2,
                              oidObserver IN VARCHAR2)
IS
BEGIN
 DELETE FROM Notification
       WHERE oid_subject = oidSubject AND
            oid_observer = oidObserver;
END;
```

FIGURA 5.9 − Procedimento *delete_notitification*

5.3 Esquema Mechanism

Para armazenar as estruturas, que mapeiam o modelo de classes do mecanismo proposto, foi criado um esquema no banco de dados Oracle, denominado Mechanism.

A seguir é apresentada a seqüência de passos para criação do Esquema Mechanism.

1. Criar *tablespace* Mechanisms_TS, usuário Mechanism, *role* mechanism_developers e privilégios (*script* anexo 1).
2. Criar as tabelas do mecanismo (*script* anexo 2).
3. Criar os tipos (*script* anexo 2).
4. Criar a especificação do pacote PassiveNotification_PAC (*script* anexo 3).
5. Criar o corpo do pacote PassiveNotification_PAC (*script* anexo 3).
6. Alterar as operações derive, promote e change_current (*script* anexo 4).
7. Criar usuários e seus privilégios (*script* anexo 6).

5.4 Exemplo de Aplicação

Um exemplo de aplicação que utiliza o mecanismo proposto é mostrado na figura 5.10. A aplicação possui a classe Veiculo, a qual representa o objeto composto e as 3 classes (Motor, Pneu e Roda), as quais representam os objetos componentes.

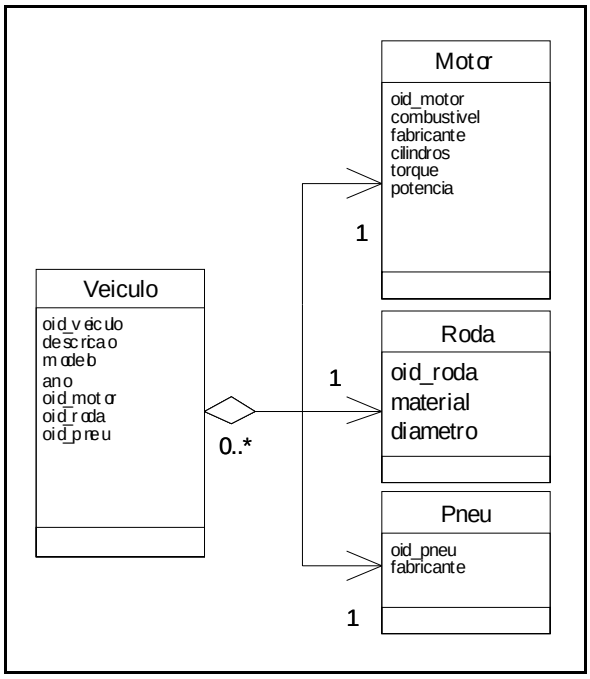

FIGURA 5.10 − Esquema Conceitual Exemplo

As classes do modelo conceitual, ilustradas na figura 5.10, são mapeadas para as tabelas da aplicação (anexo 7). Essas classes são criadas no esquema Mechanism.

O exemplo representa um ambiente de projeto de veículos, onde é importante o trabalho em grupo. Nestes contexto, consideramos a existência de dois usuários. Para que esses usuários possam subscrever eventos para notificação passiva, primeiro devem ser criadas linhas na tabela *Users*:

```
INSERT INTO Users (oid_user, users_so, user_db, e_mail)
        VALUES ('1,2,0', 'isabel', USER, 'isabefs@ig.com.br');

INSERT INTO Users (oid_user, users_so, user_db, e_mail)
        VALUES ('2,2,0', 'fonseca', USER, 'fonseca@inf.ufrgs.br');
```

O usuário '2,2,0' (fonseca) pode subscrever para notificação passiva, o objeto motor '1,1,1' para o evento *modify*, através da seguinte expressão:

```
EXECUTE NotificationPassive_PAC.subscribe_notificationp
            ('1,1,1', '2,2,0', Event_VA('modify'));.
```

Considerando que o usuário '1,2,0' altere o objeto motor '1,1,1', uma notificação é gerada pelo *trigger* referente à operação *update*:

```
UPDATE Motor
        SET potencia = '90 cv'
        WHERE oid_motor = '1,1,1';
```

O usuário que subscreveu o objeto motor '1,1,1' pode verificar as mudanças ocorridas utilizando a seguinte expressão:

```
EXECUTE NotificationPassive_PAC.consult_notification('1,1,1', '2,2,0');
```

O resultado da expressão acima é:

```
User: 1,2,0 Date: SUN 25−JUN−2000 10:58:51
Event: modify  Atributo: potencia
```

A variável de ambiente SET SERVEROUT ON deve ser ativada para que mensagens e resultados de procedimentos implementados em PL/SQL possam ser exibidos na tela.

Caso o usuário se interesse somente pela mudança ocorrida em um determinado atributo (por exemplo cilindros), a seguinte expressão pode ser usada:

```
EXECUTE NotificationPassive_PAC.subscribe_notificationp
            ('1,1,1','1,2,0', Event_VA('modify','cilindros'));
```

A seguinte expressão é executada para verificar as mudanças realizadas no objeto ’1,1,1’:

```
EXECUTE NotificationPassive_PAC.consult_notification(’1,1,1’, ’1,2,0’);
```

O resultado da expressão acima é:

```
User: 1,2,0  Date: SUN 24−SAB−2000 11:58:51
Event:modify  Atributo:cilindros
```

A expressão seguinte permite subscrever para propagação o veículo ’1,5,1’:

```
EXECUTE NotificationPassive_PAC.subscribe_propagation
        (’1,5,1’, ’1,1,1’, ’2,2,0’, Event_VA(’derive’), restricted);
```

Quando uma nova versão do objeto motor ’1,1,1’ é criada, através da operação derive_version (’1,1,1’), uma nova versão do veículo ’1,5,1’ também é criada. Para isso, a operação derive_version deve ser modificada, como descrito abaixo:

```
PROCEDURE derive_version
        (pSet_OID Set_OID, pAsc Set_OID,pDesc Set_OID, pOID OUT VARCHAR2);
Begin

/* Inserir codigo especificado em [SAG99] */

  FOR i in 1..pSet_OID.Count LOOP
/* all_observers_cur − contem todos os observers de pSet_OID */
    FOR all_observers_var IN all_observers_cur
      derive(all_observer_var);
  END LOOP;
END;
```

Os seguintes procedimentos devem ser previamente executados para que uma aplicação utilize o mecanismo:

- Inserir os usuários da aplicação na tabela Users.
- Criar as tabelas da aplicação no esquema Mechanism.
- Conectar como usuário.
- Setar o esquema Mechanism:
  (ALTER SESSION SET CURRENT_SCHEMA = mechanism).
- Criar *triggers* para as tabelas da aplicação (*script 5*).

5.5 Conclusões

Este capítulo apresentou uma proposta de implementação do mecanismo de notificação e propagação de mudanças especificado no capítulo 4.

O esquema orientado a objetos do mecanismo de notificação e propagação de mudanças foi mapeado para um esquema relacional. Todas as classes foram mapeadas para tabelas, com exceção da classe *NotificationPropagation*. As propriedades foram mapeadas para colunas. Para viabilizar a implementação algumas colunas que não correspondem a propriedades no esquema de classes foram acrescentadas.

O modelo de versões que serviu como base para este trabalho identifica cada objeto como único através do OID. Sendo assim, todas as tabelas de aplicação possuem um OID. Nas tabelas de controle do mecanismo esta coluna é chamada “oid_subject”.

A estratégia implementada foi a notificação passiva. A subscrição da notificação ativa pode ser implementada utilizando o mesmo código da notificação passiva, sendo necessário manter a informação de que tipo de notificação se trata por mail ou por mensagens na tela. O processo de envio de notificação deve ser implementado através da operação *notifya*.

6 Conclusões

Presentemente, banco de dados comercias, como ITASCA [IBE 99], OBJECTSTORE [OBJ 99] e O2 [ARD 99], possuem serviços que permitem divulgar informações de eventos ocorridos, os quais são implementados por funções disponíveis em APIs. Paralelamente, trabalhos têm buscado propor soluções para a questão da evolução de versões de objetos componentes [KAT 90, SOA 98, URT 98].

Neste trabalho, foi realizado um levantamento bibliográfico a respeito das características relevantes quanto ao gerenciamento da evolução de dados, presentes na literatura. A análise envolveu tanto banco de dados orientados a objetos, como projetos de pesquisa que tratam da questão da evolução de dados.

A evolução de dados engloba as seguintes mudanças:

- inclusão de objetos, objetos versionados e versões;
- exclusão de objetos, objetos versionados e versões;
- modificação de objetos, objetos versionados e versões.

Inicialmente foram apresentados os conceitos básicos do modelo de versões proposto por Golendziner [GOL 95]. Após, foi realizado um levantamento a respeito de procedimentos utilizados no tratamento da evolução de dados. Os diversos bancos de dados e projetos de pesquisa que possuem mecanismos para evolução de dados são analisados. Posteriormente, as soluções adotadas foram comparadas.

Dessa comparação foi constatado que nenhum dos sistemas ou propostas atende completamente todos os requisitos necessários para um amplo e abrangente suporte ao gerenciamento de evolução de dados. Alguns bancos de dados como, por exemplo, ITASCA [IBE 90] apresentam serviços de notificação ativa e passiva e não provêem propagação de mudanças. Por outro lado, projetos de pesquisa como de Katz e Oussalah disponibilizam somente soluções de propagação de mudanças.

Dentro desse contexto, é verificada a necessidade de incorporar um mecanismo completo para auxiliar a evolução de dados em banco de dados orientados a objetos, a fim de facilitar o gerenciamento da evolução de dados, auxiliar no controle da integridade destes e prover a divulgação das informações.

Este trabalho apresenta uma proposta de um mecanismo de notificação e propagação de mudanças para o modelo de versões de Golendziner. Este modelo é uma extensão do modelo de dados orientado a objetos. A proposta teve como objetivo permitir a definição e manipulação de objetos, versões e configurações, possibilitando a navegação através da hierarquia de herança e permitindo manter transparente a manipulação de versões, quando for desejado. Novas estruturas são definidas, assim como novas operações adicionais e o modelo de classes apresentado pelo modelo de versões é estendido.

O mecanismo proposto controla uma maior diversidade de eventos que os demais analisados e possui as três estratégias: notificação passiva, ativa e propagação.

Além disso, o mecanismo apresentado neste trabalho é mais completo que a princípio sugerido como trabalho futuro em [GOL 95], já que este visava a tratar somente as mudanças em versões de objetos componentes. É realizada a avaliação de outros tipos de notificação, além de mudanças nos objetos componentes.

É possível identificar as seguintes características no mecanismo proposto:

- **ortogonalidade de técnicas:** o mecanismo permite que as operações definidas sejam aplicadas tanto sobre versões, como objetos versionados e objetos não versionados;
- **seguro:** o mecanismo oferece a possibilidade de manipulação das subscrições de objetos sem adicionar concessões de alteração ou exclusão objetos;
- **possibilita a definição de escopo:** possibilidades de definir o sentido de propagação e de restringir as versões afetadas pela propagação, através do modo *restricted*;
- **flexível:** o mecanismo permite subscrever os objetos para serem monitorados, assim como cancelar a subscrição destes.

Com a utilização de mecanismos de notificação e propagação surgem alguns problemas que devem ser tratados, entre os quais estão a ambigüidade e definição do escopo de notificação e propagação. Neste trabalho são apresentadas soluções para tratar esses problemas. As operações *subscribe* e *unsubscribe* permitem ao usuário definir quais objetos devem ser monitorados. Na abordagem propagação, é possível limitar o escopo declarando a cláusula *mode* como *restricted*. e o sentido da propagação também pode ser definido.

Uma implementação do mecanismo foi apresentada para o sistema Oracle 8 e a linguagem PL/SQL foi utilizada para implementar as operações e definir a estrutura dos dados e *triggers*. Esta implementação teve como objetivo validar a estratégia notificação passiva do mecanismo proposto. Para isso foi utilizado um exemplo de projeto de veículos. Este exemplo apresenta um ambiente de projetos onde a interação entre usuários é relevante.

É possível implementar as operações *notifyp* e *propagate* com um número menor de parâmetros, como mostrado na seção 5.2.2, do que a princípio definido nas seções 4.3.10 e 4.3.11. Optou−se por manter na especificação da operação todos os parâmetros porque assim ficam mais evidentes para o projetista da implementação quais informações são utilizadas.

Modelos de banco de dados orientados a objetos para ambientes de projeto e mecanismos que tratam a evolução de dados [KHO 95] são utilizados para construção de *groupware*. Sendo assim, o modelo de Golendziner juntamente com o mecanismo de notificação e propagação de mudanças pode ser avaliado para isto.

A notificação e propagação de mudanças apresentada neste trabalho visou tratar somente a evolução de dados. Um mecanismo para tratar a evolução de esquemas, como por exemplo a inclusão de atributos e classes para o modelo de versões de Golendziner pode ser explorado em trabalhos futuros.

As necessidades das aplicações de projeto, para as quais esse trabalho visa a satisfazer, requerem maior complexidade na estrutura e nas alterações, não requerendo grande volume de dados. Justifica−se, portanto, a inserção do mecanismo proposto, aparentemente complexo, e com isso novas estruturas e operações no modelo. Além disso, somente os objetos que o usuário determina são gerenciados pelo mecanismo.

Anexo 1 *Script* para criação do Esquema Mechanism

```
/* ____________________________________________________________
                    Esquema Mechanism
   ____________________________________________________________
*/

CREATE TABLESPACE mechanisms_ts
  DATAFILE 'c:\oracle\Oradata\orcl\arquivos\mechanisms.dat' SIZE 20M
  DEFAULT STORAGE (INITIAL 10K NEXT 50K
            MINEXTENTS 1 MAXEXTENTS 999);

CREATE TABLESPACE temp_ts
 DATAFILE 'c:\oracle\Oradata\orcl\arquivos\temp.dat' SIZE 20M
  DEFAULT STORAGE (INITIAL 10K NEXT 50K
            MINEXTENTS 1 MAXEXTENTS 999);

CREATE USER mechanism
  IDENTIFIED BY developer
  DEFAULT TABLESPACE mechanisms_ts
  QUOTA 5M ON mechanisms_ts
  QUOTA 1M ON temp_ts
  QUOTA 1M ON system;

CREATE ROLE mechanism_developers IDENTIFIED by developer;

/*
                    Atribui previlegios ao role criado acima
*/

GRANT "CONNECT" TO mechanism_developers;
GRANT CREATE SESSION TO mechanism_developers;
GRANT CREATE TABLE TO mechanism_developers;
GRANT CREATE VIEW TO mechanism_developers;
GRANT CREATE PROCEDURE TO mechanism_developers;
GRANT CREATE TYPE TO mechanism_developers;
GRANT CREATE TRIGGER TO mechanism_developers;
GRANT DROP ANY TABLE TO mechanism_developers;
GRANT DROP ANY VIEW TO mechanism_developers;
GRANT DROP ANY PROCEDURE TO mechanism_developers;
GRANT DROP ANY TYPE TO mechanism_developers;


/*
                    Atribui previlegios do role ao usuario mechanism
*/
GRANT mechanism_developers TO mechanism;

COMMIT;
```

Anexo 2 *Script* para criação das tabelas do esquema Mechanism

```
/* _____________________________________________________________

    Tabelas e tipos do Mecanismo de Notificacao e Propagacao

  _____________________________________________________________

*/


CREATE OR REPLACE TYPE Event_VA AS VARRAY(2) OF VARCHAR2(30);

CREATE TABLE Users (
                        oid_user VARCHAR2(30) PRIMARY KEY,
                        users_so VARCHAR2(80) NOT NULL,
                        user_db VARCHAR2(80) NOT NULL,
                        e_mail VARCHAR2(80) NOT NULL);


CREATE TABLE NotificationSubscription (
                        oid_subject VARCHAR2(30) NOT NULL,
                        oid_observer VARCHAR2(30) NOT NULL,
                        kind VARCHAR2(1) NOT NULL,
                        event VARCHAR2(30) NOT NULL,
                        attr_modify VARCHAR2(30),
                        PRIMARY KEY (oid_subject, oid_observer, kind, event),
                        FOREIGN KEY (oid_observer) REFERENCES Users(oid_user));


CREATE TABLE Notification (
                        oid_subject VARCHAR2(30) NOT NULL,
                        oid_user VARCHAR2(30) NOT NULL,
                        oid_observer VARCHAR2(30) NOT NULL,
                        event VARCHAR2(30) NOT NULL,
                        attr_modify VARCHAR2(30),
                        kind VARCHAR2(1) NOT NULL,
                        timestamp DATE NOT NULL,
                        PRIMARY KEY (oid_subject, oid_user, oid_observer, event, timestamp),
                        FOREIGN KEY (oid_user) REFERENCES Users(oid_user),
                        FOREIGN KEY (oid_observer) REFERENCES Users(oid_user));


CREATE TABLE PropagationSubscription (
                        oid_subject VARCHAR2(30) NOT NULL,
                        oid_observer VARCHAR2(30) NOT NULL,
                        event VARCHAR2(30) NOT NULL,
                        modo VARCHAR2(10),
                        PRIMARY KEY (oid_subject, oid_observer, event, modo));
```

Anexo 3 *Script* para criação do pacote NotificationPassive_PAC

```
/* ____________________________________________________________

            Pacote NotificationPassive_PAC

 ____________________________________________________________
*/
/* Criar primeiro (separado) o pacote e depois o corpo do pacote*/

CREATE OR REPLACE
PACKAGE NotificationPassive_PAC
AS
 PROCEDURE subscribe_notificationp(oidSubject IN VARCHAR2,
                                   oidObserver IN VARCHAR2,
                                   ev IN Event_VA);

 PROCEDURE unsubscribe_notificationp(oidSubject IN VARCHAR2,
                                     oidObserver IN VARCHAR2,
                                     ev IN Event_VA);
 PROCEDURE consult_notification(oidSubject IN VARCHAR2,
                                oidObserver IN VARCHAR2);

 PROCEDURE delete_notification(oidSubject IN VARCHAR2,
                               oidObserver IN VARCHAR2);


 PROCEDURE notifyp(oidSubj IN VARCHAR2,
                   oidObserv IN VARCHAR2,
                   oidUser IN VARCHAR2,
                   ev IN Event_VA,
                   timestamp_var IN DATE);

 PROCEDURE notifyp_all_observers(oidSubj IN VARCHAR2,
                                 ev IN Event_VA,
                                 timestamp_var IN DATE);
END NotificationPassive_PAC;


/* Body */

CREATE OR REPLACE
PACKAGE BODY NotificationPassive_PAC
AS

 PROCEDURE subscribe_notificationp(oidSubject IN VARCHAR2,
                                   oidObserver IN VARCHAR2,
                                   ev IN Event_VA)
 IS
 BEGIN
            IF ev.COUNT > 1 THEN
                INSERT INTO NotificationSubscription (oid_subject, oid_observer, kind,
            event, attr_modify)
                    VALUES (oidSubject, oidObserver,'p', ev(1), ev(2));
            ELSE
                INSERT INTO NotificationSubscription (oid_subject, oid_observer, kind,
            event)
                    VALUES (oidSubject, oidObserver,'p', ev(1));
            END IF;

 END;
```

```
PROCEDURE unsubscribe_notificationp(oidSubject IN VARCHAR2,
                                    oidObserver IN VARCHAR2,
                                    ev IN Event_VA)
IS
BEGIN
 DELETE FROM NotificationSubscription
                WHERE oid_subject = oidSubject AND
                    oid_observer = oidObserver AND
                    kind = 'p' AND
                    event = ev(1);

END;


PROCEDURE consult_notification(oidSubject IN VARCHAR2,
                               oidObserver IN VARCHAR2)
IS

CURSOR all_notifications_cur RETURN Notification%ROWTYPE
                IS
  SELECT * FROM Notification
  WHERE oid_subject = oidSubject AND
                    oid_observer = oidObserver;

BEGIN
                FOR all_notifications_var IN all_notifications_cur
                 LOOP
                        DBMS_OUTPUT.PUT('User:'  || all_notifications_var.OID_USER);
                        DBMS_OUTPUT.PUT('                                Date:'          ||
                TO_CHAR(all_notifications_var.timestamp,
                                                    'DY DD-MON-YYYY HH24:MI:SS'));
                        DBMS_OUTPUT.PUT('  Event:' || all_notifications_var.event);
                        IF (all_notifications_var.event = 'modify')
                        THEN
                                DBMS_OUTPUT.PUT_LINE('                Atributo:'        ||
                all_notifications_var.attr_modify);
                        ELSE
                                DBMS_OUTPUT.PUT_LINE('');
                        END IF;
                END LOOP;
END;


PROCEDURE delete_notification(oidSubject IN VARCHAR2,
                              oidObserver IN VARCHAR2)
IS
BEGIN
 DELETE FROM Notification
                WHERE oid_subject = oidSubject AND
                    oid_observer = oidObserver;
END;

/*
                Insere uma notificacao (uma linha na tabela Notification).
                Um valor pode ser inserido no atributo 'attr_modify'
                quando o evento é 'modify'.
*/
```

```
PROCEDURE notifyp(oidSubj IN VARCHAR2,
                                                oidObserv IN VARCHAR2,
                                                oidUser IN VARCHAR2,
                                                ev IN Event_VA,
                                                timestamp_var IN DATE)
IS

BEGIN
                        IF ev.COUNT > 1 THEN
                                INSERT INTO Notification
                                (oid_subject, oid_user, oid_observer, event, attr_modify, kind, timestamp)
                                VALUES
                           (oidSubj,oidUser,oidObserv, ev(1), ev(2),'p', timestamp_var);
                        ELSE
                                INSERT INTO Notification
                                (oid_subject, oid_user, oid_observer, event, kind,timestamp)
                                VALUES
                           (oidSubj, oidUser, oidObserv, ev(1), 'p', timestamp_var);
                        END IF;
END;


/*
   É chamado pelos trigger para delete e pelas operacoes
   promote, derive, change_current,menos pela modify
*/

PROCEDURE notifyp_all_observers(oidSubj IN VARCHAR2,
                                                        ev IN Event_VA,
                                        timestamp_var IN DATE)
IS

CURSOR all_observers_cur
                        IS
                        SELECT oid_observer
                        FROM   NotificationSubscription
                        WHERE  oid_subject = oidSubj AND
                           kind        = 'p' AND
                           event       = ev(1);

oid_user_var VARCHAR2(30);

BEGIN
                        SELECT oid_user INTO oid_user_var FROM Users WHERE user_db = USER;

                        FOR all_observers_var IN all_observers_cur LOOP
                                NotificationPassive_PAC.notifyp(oidSubj,all_observers_var.oid_observer,
                                                                                                                                oid_
                        user_var, ev, timestamp_var);
                        END LOOP;

END;

END NotificationPassive_PAC;
```

Anexo 4 *Script* das operações *change_current, promote* e *derive*

```
/* _____________________________________________________________

      Operacoes do modelo de versoes (change_current, promote, derive)
   _____________________________________________________________
*/


/* Altera a versao corrente e permite notificacao */

PROCEDURE change_current(Self IN VersionedObject, ObjVers VARCHAR2, Versao VARCHAR2)

Begin

/* Inserir codigo especificado em [SAG99] */

/* Esta procedure permite inserir uma linha na tabela Notification
   com o ObjVers (objeto versionado) para o evento change */

                        NotificationPassive_PAC.notifyp(ObjVers,'change');
END;


/* Promove uma ou mais versoes e permite a notificacao */

PROCEDURE promote(Self IN Version, pOID IN VARCHAR2, AllAsc Boolean, AllRef Boolean)

Begin

/* Inserir codigo especificado em [SAG99] */

/* Insere uma linha na tabela Notification
   com a versao (pOID) para o evento promote */

                        NotificationPassive_PAC.notifyp_all_observers(pOID,'promote');

/* Insere uma ou mais linhas na tabela Notification com a(s) versao(s)
    ascendentes (OIDAsc da versao pOIDs para o evento promote */
                        LOOP ...
                                NotificationPassive_PAC.notifyp_all_observers(OIDAsc,'promote');
                        END LOOP;

/* Insere uma ou mais linhas na tabela Notification          com a(s) versao(s)
   (OIDVer) que fazem referencia a versao pOI para o evento promote
*/
                        LOOP ....
                                NotificationPassive_PAC.notifyp_all_observers(OIDVer,'promote');
                        END LOOP;
END;


/* Cria uma nova versao que permite a notificacao */

PROCEDURE derive_version(pSet_OID Set_OID, pAsc Set_OID,
        pDesc Set_OID, pOID OUT VARCHAR2);


Begin

/* Inserir codigo especificado em [SAG99] */
```

```
                    FOR i in 1..pSet_OID.Count LOOP
                    /* Count é uma funcao que retorna o numero total de
                       elementos da tabela.
                       Todos os observers de cada versao do conjunto pSet_OID recebem
                       uma notificacao com o evento derive.
                    */
                    NotificationPassive_PAC.notifyp_all_observers(pSet_OID(i),'derive');
                    END LOOP;
END;
```

Anexo 5 *Script* para criação de *triggers*

```
/* ____________________________________________________________
      Exemplo de triggers os quais devem criados em todas as
      tabelas da aplicacao para que a notificacao seja possivel
____________________________________________________________
*/

/* Criar os triggers separados */

CREATE OR REPLACE TRIGGER modifymotor
                        AFTER UPDATE ON Motor FOR EACH ROW

DECLARE
                        CURSOR all_observers_cur
                        IS
                        SELECT oid_observer, attr_modify
                        FROM   NotificationSubscription
                        WHERE  oid_subject = :new.oid_motor AND
                           kind       = 'p' AND
                           event      = 'modify';

                        oid_user_var VARCHAR2(30);

BEGIN
                        SELECT oid_user INTO oid_user_var FROM Users WHERE user_db = USER;
                        FOR all_observers_var IN all_observers_cur LOOP
                        /* Cada IFs possibilita subscricao de um atributo */
                                IF ((:old.cilindros != :new.cilindros) AND
                                   ((all_observers_var.attr_modify IS NULL) OR
                                    (all_observers_var.attr_modify = 'cilindros')
                                   )
                                  )
                                THEN
                                        NotificationPassive_PAC.notifyp(:old.oid_motor,all_observers_var.
                        oid_observer,

                                oid_user_var, Event_VA('modify','cilindros'), sysdate);
                                ELSE

                                        IF ((:old.potencia != :new.potencia) AND
                                           ((all_observers_var.attr_modify IS NULL) OR
                                            (all_observers_var.attr_modify = 'potencia')
                                           )
                                          )
                                        THEN
                                        NotificationPassive_PAC.notifyp(:old.oid_motor,all_observers_var.
                        oid_observer,

                                oid_user_var, Event_VA('modify','potencia'), sysdate);
                                        END IF;

                                END IF;
                        END LOOP;
END modifymotor;
```

```
/* trigger DELETE */
CREATE OR REPLACE TRIGGER deletemotor
                        AFTER DELETE ON Motor FOR EACH ROW
BEGIN
   NotificationPassive_PAC.notifyp_all_observers(:old.oid_motor,Event_VA('delete'), sysdate);
END deletemotor;
```

Anexo 6 *Script* para criação de usuários e concessão de privilégios

```
/* ____________________________________________________________
        Esquema Isabel e Fonseca
   ____________________________________________________________
*/

CREATE USER isabel
  IDENTIFIED BY isa
  DEFAULT TABLESPACE mechanisms_ts
  QUOTA 5M ON mechanisms_ts
  QUOTA 1M ON temp_ts
  QUOTA 1M ON system;

CREATE USER fonseca
  IDENTIFIED BY ana
  DEFAULT TABLESPACE mechanisms_ts
  QUOTA 5M ON mechanisms_ts
  QUOTA 1M ON temp_ts
  QUOTA 1M ON system;


CREATE ROLE application_users;

GRANT CONNECT TO application_users;
GRANT CREATE SESSION TO application_users;
GRANT CREATE TABLE TO application_users;
GRANT CREATE VIEW TO application_users;
GRANT CREATE TYPE TO application_users;
GRANT CREATE TRIGGER TO application_users;

GRANT application_users TO fonseca;
/* user deve ser ADM */
GRANT EXECUTE type Event_VA TO fonseca;
GRANT SELECT, INSERT ON Users TO fonseca;
GRANT EXECUTE ON NotificationPassive_PAC TO fonseca;
GRANT EXECUTE ON NotificationPassive_PAC TO fonseca;
GRANT SELECT, INSERT, DELETE, UPDATE ON Motor TO fonseca;
GRANT SELECT, INSERT, DELETE, UPDATE ON Pneu TO fonseca;
GRANT SELECT, INSERT, DELETE, UPDATE ON Roda TO fonseca;
GRANT SELECT, INSERT, DELETE, UPDATE ON Veiculo TO fonseca;

GRANT application_users TO isabel;
/* user deve ser ADM */
GRANT EXECUTE ANY TYPE TO isabel;
GRANT SELECT, INSERT ON Users TO isabel;
GRANT EXECUTE ON NotificationPassive_PAC TO isabel;
GRANT EXECUTE ON NotificationPassive_PAC TO isabel;
GRANT SELECT, INSERT, DELETE, UPDATE ON Motor TO isabel;
GRANT SELECT, INSERT, DELETE, UPDATE ON Pneu TO isabel;
GRANT SELECT, INSERT, DELETE, UPDATE ON Roda TO isabel;
GRANT SELECT, INSERT, DELETE, UPDATE ON Veiculo TO isabel;

COMMIT;
```

Anexo 7 *Script* para criação das tabelas do Exemplo

```
/* ____________________________________________________________
                    Tabelas da Aplicacao Exemplo
   ____________________________________________________________
*/


/* Motor */
CREATE TABLE Motor (
                    oid_motor VARCHAR2(30) PRIMARY KEY,
                    combustivel VARCHAR2(20),
                    fabricante VARCHAR2(40),
                    cilindros VARCHAR2(20),
                    torque VARCHAR2(20),
                    potencia VARCHAR2(20));

/* Roda */
CREATE TABLE Roda (
                    oid_roda VARCHAR2(30) PRIMARY KEY,
                    material VARCHAR2(20),
                    diametro VARCHAR2(10));

/* Pneu */
CREATE TABLE Pneu (
                    oid_pneu VARCHAR2(30) PRIMARY KEY,
                    fabricante VARCHAR2(20));

/* Veiculo */
CREATE TABLE Veiculo (
                    oid_veiculo VARCHAR2(30) PRIMARY KEY,
                    descricao VARCHAR2(30),
                    modelo VARCHAR2(40),
                    ano DATE,
                    oid_motor VARCHAR2(30),
                    oid_roda VARCHAR2(30),
                    oid_pneu VARCHAR2(30),
                    FOREIGN KEY (oid_motor) REFERENCES Motor(oid_motor),
                    FOREIGN KEY (oid_roda) REFERENCES Roda(oid_roda),
                    FOREIGN KEY (oid_pneu) REFERENCES Pneu(oid_pneu));

COMMIT;
```

Bibliografia

[ALL 97] ALLOUI, Ilham. **ItascaFlow:** An Object Database Technology for Supporting Distributed and Adaptive Workflow Systems. Disponível por www em http://www.ibex.ch (14 jun. 1999).

[ARD 99] ARDENT SOFTWARE. **O2 Object Database System.** Disponível por www em http://www.ardentsoftware.com (20 jun. 1999).

[ARD 99a] ARDENT SOFTWARE. **O2Notification.** Disponível por www em http://www.ardentsoftware.com (20 jun. 1999).

[ATW 85] ATWOOD, T. An object−oriented DBMS for design support applications. In: COMPINT. 1985, Montreal, Canada. **Proceedings**... [S.l.:s.n.], 1985. p.299−307.

[BUS 96] BUSCHAMANN, F. **Pattern−Oriented Software Arquitecture:** A System of Patterns. England: John Wiley, 1996. 457p.

[BOR 95] BORGES, M. Using Database Versions to Support Awareness in Group Interactions. In: ECSCW WORKSHOP ON VERSION CONTROL, 1995, Sweden. **Proceedings...** Stockholm: [s.n.], 1995.

[CHO 86] CHOU, H. ; KIM, W. A Unifying Framework for Version Control in a CAD nvironment. In**:** INTERNATIONAL CONFERENCE ON VERY LARGE DATA BASES, 12., 1986, Kyoto. **Proceedings...** Austin: VLDB, 1986. p.336−344.

[CHO 88] CHOU, H. T; KIM, W. Version and Change notification in an object− oriented database system**.** In: DESIGN AUTOMATION CONFERENCE, 25., 1988, Anahein. **Proceedings**… New York: ACM, 1988. 730p. p.275−281.

[ELM94] ELMASRI, Ramez; NAVATHE, Shamkant, B. **Fundamentals of Database Systems.** Redwood: Benjamin/Cummings, 1994.

[FON 98] FONSECA, Ana C. G**. Estudo de mecanismos que tratam a evolução de versões de objetos:** Trabalho individual. Porto Alegre: CPGCC da UFRGS. 1998 48p.

[FUR 98] FURLAN, J. D. **Modelagem de Objetos através da UML.** São Paulo: Makron Books, 1998. 329p.

[GAL 98] GALANTE, R. **Um Modelo de Evolução de Esquemas Conceituais para Banco de Dados Orientados a Objetos como o Emprego de Versões.** Porto Alegre: CPGCC da UFRGS, 1998. 92p. Dissertação de Mestrado.

[GAM 99] GAMMA, F. **Padrões de Projetos:** Soluções Reutilizáveis de Software Orientado a Objetos. Porto Alegre: Bookman, 1999. 364p.

[GAR 99] GARLAND, J.; ANTHONY, D. **Using Objectivity on the IRIDIUM System.** Disponível por www em http://web.cs.city.ac.uk/homes/akmal/papers.dir/97−oopsla−expanded.html (22 set. 1999).

[GOL 95] GOLENDZINER, Lia Goldstein. **Um modelo de versões para banco de dados orientados a objetos**. Porto Alegre: CPGCC da UFRGS, 1995. 147p. Tese de doutorado.

[GOL 95a] GOLENDZINER, Lia Goldstein; SANTOS, Clesio Saraiva dos. Definição e manipulação de versões em banco de dados orientados a objetos. In: SIMPÓSIO BRASILEIRO DE BANCO DE DADOS, 10., 1995, Recife−PE. **Anais...** Recife:UFPE/DI, 1995. p. 335−349.

[GOL 95b] GOLENDZINER, Lia Goldstein; SANTOS, Clesio Saraiva dos. Uma abordagem multi−nível para suporte a versões em banco de dados orientados a objetos. In: CONGRESSO DA SOCIEDADE BRASILEIRA DE COMPUTAÇÃO, 15., 1995, Canela−RS. **Anais...** Porto Alegre: SBC,1995. p. 1127−1138.

[GOL 95c] GOLENDZINER, Lia Goldstein; SANTOS, Clesio Saraiva dos. Versions and configurations in object−oriented database systems: a uniform treatment. In: ACM SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, 7., 1995, Pune, Índia. **Proceedings...** New Delhi: Mcgraw−Hill, 1995. p. 18−37.

[GOL 97] GOLENDZINER, L. G; SANTOS, C.S. Versions and configurations in object−oriented concepts. In: DATABASE SYSTEMS FOR ADVANCED APPLICATIONS (DASFAA), 5.,1997, Melbourne, Australia. **Proceedings...** [S.1:s.n], 1997. p. 115−124.

[GOK 98] GOKMAN, Mark; INGRAM, Jonathan. **Oracle8 PL/SQL Black Book.** Arizona: Coriolis Group Boks, 1998. 694 p.

[IBE 99] IBEX OBJECT SYSTEMS. **Distributed Object Database Management System**. Technical Summary Release 2.3.5. Disponível por www em http://www.ibex.ch/techsum/index.htm (14 jun. 1999).

[KAT 87] KATZ, R. H.; CHANG, E. Managing change in computer−aided design database. In: CONFERENCE ON VERY LARGE DATA BASES, 13.,1987, Brighton. **Proceedings...** England: VLDB, 1987.

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

[KHO 95] KHOSHAFIAN, S. **Introduction to Groupware, Workflow, and Workgroup Computing.** New York: John Wiley & Sons, 1995. 376p.

[LAU 98] LAUSEN, G.; VOSSEN, G. **Models and Languages of Object−Oriented Databases.** Harlow: Addison−Wesley, 1998. 218 p.

[LIM 92] LIMA, M. J. ; CAVALCANTI, M. R. Tratamento de Versões em Documentos Multimídia. In: SIMPÓSIO BRASILEIRO DE BANCO DE DADOS, 7., 1992, Porto Alegre. **Anais . . .** Porto Alegre: SBC, 1992.

[LIM 96] LIMA, G. **Gerenciamento de transações**: Um estudo e uma proposta. Campinas: [s.n.], 1996. 85 p. Dissertação de Mestrado.

[LIM 97] LIMA, G.; TOLEDO, M. Um modelo de transações cooperativas integrado a um modelo de versões. In: SIMPÓSIO BRASILEIRO DE BANCO DE DADOS, 12., 1997. **Anais..** [S.l.: s.n], 1997.

[LIN 99] LINKOPING UNIVERSITY. **LINCKS Research Database System**. Disponível por www em http://www.ida.liu.se/~lincks/.(1999).

[O2T 96] O2 TECHNOLOGY. **Version Management.** Reference Manual. Versailles: O2 Technology, 1996.

[OBJ 99] OBJECT DESIGN. **ObjectStore Enterprise Edition.** Disponível por www em fttp://www.odi.com/ (14 ago. 1999).

[OBJ 99a] OBJECTSTORE. **ObjectStore Java API User Guide**. Disponível por www em http://galileo.ethz.ch/ODI/OSJI/doc/apiug/6t_notif.htm (14 set. 1999).

[OUS 96] OUSSALAH, C.; URTADO, C. Adding Semantics for Version Propagation in OODBs. In: WORKSHOP ON DATABASES AND EXPERT SYSTEMS APPLICATIONS, 1996, Switzerland. **Proceedings...** Zurich: IEEE Computer Society Press, 1996.

[OUS 96a] OUSSALAH, C.; URTADO, C. Semantic Rules to Propagate Versions in Object−Oriented Databases. In**:** WORKSHOP OF THE MOSCOU ACM SIGMOD CHAPTER, ADVANCES IN DATABASE AND INFORMATION SYSTEMS, 3., 1996, Russia. **Proceedings...** Moscou: ACM SIGMOD, 1996.

[OUS 97] OUSSALAH, C.; URTADO, C. Complex Object Versioning. In**:** INTERNATIONAL CONFERENCE ON ADVANCED INFORMATION SYSTEMS ENGINEERING, 1997, Spain. **Proceedings...** Barcelona : Computer Society , 1997. p 259−272.

[NOR 98] NORONHA, M. **Uma proposta de suporte a versões em documentos estruturados.** Porto Alegre: CPGCC da UFRGS, 1998. 78p. Dissertação de Mestrado.

[ROM 00] ROMA, A. **Um Modelo para Evolução de Esquemas em Bancos de Dados Orientados a Objetos Utilizando Versões e Operações Complexas**. Porto Alegre: CPGCC da UFRGS, 2000. 97p. Dissertação de Mestrado.

[ROS 99] ROSTOCK UNIVERSITY.**ObjectStore System**. Disponível por www em http://wwwdb.informatik.uni−rostock.de/~jo/ostore5/user1/index.htm (23 jun. 1999).

[SAG 99] SAGGIORATO, Sílvia Maria. **Mapeamento de Esquemas Orientados a Objetos com Versões para Esquemas Objeto−Relacionais.** Porto Alegre: PPGC da UFRGS, 1999. 112p. Dissertação de Mestrado.

[SKI 99] SKIADELLI, Maria. **Object Oriented database system evaluation for the DAQ system**. Tese de Doutorado. Disponível em http://rd13doc.cern.ch/public/doc/Note108/thesis.book.html (14mai. 1999).

[SOA 95] SOARES, L. G.; RODRIGUES, R. F; CASANOVA, M. A. et. al. Nested composite nodes and version control in an open hypermedia system. **Information Systems**, Special issue on Multimedia Information Systems, New York, v. 20, n. 6, p. 501−591, 1995.

[SOA 98] SOARES, L. G.; SOUZA, G. L. S.; RODRIGUES, R. F. et. al. Propagação e Proliferação de Versões em Modelos Conceituais Hipermídia com Composições. In: SIMPÓSIO BRASILEIRO DE BANCO DE DADOS, 13., 1998. **Anais ...** Maringá: SBC, 1998.

[TAL 93] TALES, G.; OUSSALAH, C.; COLINAS, M. F**.** Versions and Simple and composite Objects. In: VLDB CONFERENCE, 19. , 1993, Dublin. **Proceedings...** Dublin: VLDB, 1993. p.62−72.

[TOL 99] TOLEDO, M. B.Supporting Cooperation with Integrated Transaction and Session Facilities. In: SIMPÓSIO BRASILEIRO DE BANCO DE DADOS, 1999. **Anais...** Florianópolis: [s.n.],1999.

[URT 98] URTADO, C.; OUSSALAH, C. Complex entity versioning at two granularity levels. **Information Systems**, Elmsford, v.23, n.3/4, p. 196−216, May−June 1998.

[VER 99] VERSANT OBJECT TECHNOLOGY. **The Database for Objects**. Disponível por www em http://www.versant.com (10 maio 1999).

[VER 99a] VERSANT OBJECT TECHNOLOGY. **Publish und Subscribe**. Disponível por www em http://www.versant.com/us/products/release/index.html.