ATUALIZAÇÃO DE BANCOS DE DADOS OBJETO-RELACIONAIS ATRAVÉS DE VISÕES XML

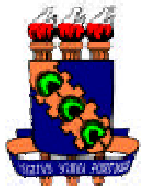
Por

Wamberg Gláucon Chaves de Oliveira

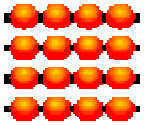
Orientadora: Profª. Drª.Vânia Maria Ponte Vidal

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

FORTALEZA - CEARÁ
Setembro - 2004



UNIVERSIDADE FEDERAL DO CEARÁ
DEPARTAMENTO DE COMPUTAÇÃO
MESTRADO EM CIÊNCIA DA COMPUTAÇÃO



ATUALIZANDO BANCO DE DADOS OBJETO-RELACIONAL ATRAVÉS DE VISÕES XML

Wamberg Gláucon Chaves de Oliveira[1]

Setembro de 2004

Banca Examinadora

- Profa. Dra. Vânia Maria Ponte Vidal
- Prof. Dr. Alberto Henrique Frade Laender
- Prof. Dr. Javam de Castro Machado
- Profa. Dra. Bernadette Farias Lóscio

[1] Bolsista CAPES

Agradecimentos

De formas variadas e em momentos diferentes, muitas pessoas direta ou indiretamente, ajudaram no desenvolvimento deste trabalho. A cada uma delas, sinceramente, agradeço.

À Deus que me guia em todos os meus passos e a Ele entrego minha vida.

À minha mãe, que sempre foi meu amparo, porto seguro, durante todos estes anos da minha vida e que fez de tudo para que eu trilhasse pelo melhor caminho.

Aos meus irmãos Elker e Maysa, por terem aguentado o meu mau humor nos momentos difíceis e por compartilharem dos momentos de felicidade.

À minha orientadora, Prof.ª Vânia Vidal, pela orientação criteriosa, por seu entusiasmo, paciência, atenção e participação na realização deste trabalho.

À Prof.ª Bernadete Farias Lóscio por disponibilizar seu tempo e me ajudar na redação final. da dissertação.

Ao Departamento de Computação e a CAPES por tornarem possível a realização deste mestrado.

Aos amigos do laboratório de banco de dados, presentes ou não, Lineu,Valdiana, Marcel, Juliana, Renata, Fernando, Denis, Fábio, que de alguma forma me ajudaram muito, pelo apoio e pela convivência agradável que irei sentir falta.

Aos colegas do mestrado que durante o curso foram sempre amigos e prestativos, e que de alguma forma tornaram a vida acadêmica menos árdua.

À Vanessa, meu amor e grande amiga, que sempre esteve do meu lado nos momentos difíceis, com carinho e alegria, e me ensinou a persistir nos meus sonhos. Obrigado por existir na minha vida.

Resumo

O modelo XML (e**X**tended **M**arkup **L**anguage) vem se tornando um padrão para troca e integração de dados na WEB. Isto cria a necessidade de publicar dados, que estão armazenados em bancos de dados convencionais, no formato XML. A publicação desses dados é feita através do uso de visões XML. Atualmente, diversos sistemas têm proposto soluções para consultar bancos de dados convencionais utilizando visões XML. Entretanto, com o intuito de tornar XML um padrão completo para representação de dados e compartilhamento de informações, deve ser possível não somente consultar, mas também atualizar esses bancos de dados através de visões XML.

Neste trabalho abordamos o problema de atualização de bancos de dados objeto-relacionais (BDOR) através de visões XML. Uma atualização da visão é simplesmente uma atualização que é especificada em uma visão XML, mas que deve ser traduzida em atualizações a serem executadas no banco de dados. O problema de tradução da atualização da visão XML refere-se à questão de definir traduções corretas para atualizações submetidas através de visões XML.

No enfoque proposto, utilizamos o XML *Publisher*, um *framework* para publicação de dados armazenados em bancos de dados relacionais ou objeto-relacionais no formato XML. O *framework* permite a publicação de visões XML, assim, utilizamos uma extensão da linguagem *XQuery*, também apresentada neste trabalho, para atualizar o banco de dados através dessas visões.

No enfoque proposto, para atualizar o banco de dados através das visões XML, deve-se definir uma visão de objetos, denominada visão de objetos *default* (VOD), de modo que os objetos da VOD possuam a mesma estrutura dos elementos da visão XML. Atualizações definidas sobre o esquema da visão XML são traduzidas em atualizações sobre o esquema da respectiva VOD, e que por sua vez são traduzidas, através de tradutores de atualização (triggers) em atualizações sobre o banco de dados.

Para o XML *Publisher* desenvolveu-se um ambiente para facilitar a publicação e manutenção das visões XML publicadas. No ambiente proposto, a publicação de uma visão começa com o projetista definindo, através de uma interface gráfica, o esquema XML da

visão. Em seguida, o esquema da VOD é automaticamente gerado pela ferramenta DSG (***D****efault Object View* ***S****chema* ***G****enerator*). Então, a partir do esquema da VOD gerado e do esquema do banco de dados, o projetista utiliza a ferramenta TAV (Tradutores de Atualização de Visão) para gerar de forma gráfica as assertivas de correspondência da visão, as quais especificam os relacionamentos entre o esquema da visão e o esquema do banco de dados. Com base nas assertivas da visão, TAV gera automaticamente os tradutores de atualização (*instead of triggers*) para a VOD.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1 - Introdução

1.1 Motivação

Desde seu surgimento, XML [1] vem rapidamente emergindo como padrão para troca, publicação e integração de dados na Web, pois é uma linguagem para representação de dados autodescritível e bastante flexível, o que a torna ideal para representar dados estruturados e semi-estruturados.

De fato, aplicações que necessitam processar dados representados em XML estão sendo desenvolvidas. Como exemplo, podemos citar as aplicações de comércio eletrônico, onde diferentes organizações colaboram para atender um cliente, bem como aplicações direcionadas aos dispositivos portáteis.

Além do mais, como a maioria dos dados está armazenada em bancos de dados convencionais, isto é, relacionais ou baseados em objetos, isto cria a necessidade de se publicar esses dados corporativos no formato XML. Uma forma geral de publicar esses dados é feita através do uso de visões XML, que podem ser consultadas e atualizadas por aplicações/usuários *web* utilizando linguagens de consultas próprias para XML, tais como *XML*-QL [2], *XPath* [3], *XQuery* [4] etc. Sendo assim, temos que XML está sendo usado como uma camada intermediária entre aplicações e bancos de dados.

Já existem alguns sistemas como o *XPeranto* [5,6,7] e o *Silkroute* [8,9] que dão suporte à publicação de dados relacionais, através de visões XML. Nesses sistemas, consultas são submetidas às visões XML e traduzidas em consultas nos bancos de dados. Entretanto, estes sistemas não efetuam atualizações sobre visões XML. Objetivando tornar XML um padrão completo para representação de dados, compartilhamento e integração de informações, deve-se permitir não somente consultar, mas também possibilitar que usuários e componentes de aplicações atualizem estes bancos de dados através de visões XML.

O uso de visões XML também tem uma grande importância em sistemas de integração de dados que adotam uma arquitetura de mediadores baseada em XML [10,11,12,13]. O mediador suporta uma visão integrada XML e as fontes locais exportam visões XML. Consultas e atualizações submetidas ao mediador são decompostas em

consultas e atualizações sobre as visões XML das fontes locais. Consultas e atualizações sobre as visões XML são então traduzidas em consultas e/ou atualizações numa linguagem específica da fonte de dados correspondente.

1.2 Problema de Atualização de Visão

Na arquitetura de três níveis de esquemas, as visões constituem interfaces através das quais os usuários acessam e atualizam um banco de dados. Consultas e atualizações especificadas nas visões devem ser traduzidas em consultas e atualizações a serem executadas no banco de dados. O processo de tradução de uma atualização de visão é descrito no diagrama da Figura 1.1. O estado inicial do banco de dados σ_s é mapeado pelo mapeador de instâncias DEF_V no estado da visão σ_v. O usuário especifica a atualização μ sobre o estado da visão. A atualização da visão μ deve ser traduzida em uma seqüência de atualizações sobre o banco de dados $\tau(\mu)$. $\tau(\mu)$ é executada no estado do banco de dados σ_s para obter o novo estado do banco de dados σ'_s. Com o novo estado do banco de dados σ'_s, obtemos o novo estado da visão correspondente σ'_v [14].

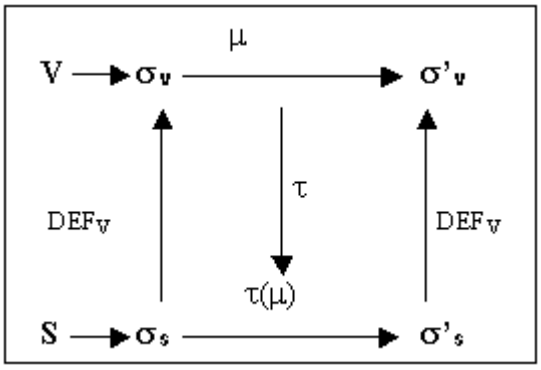


Figura 1.1: Tradução de atualização de visão

Um pedido de atualização submetido à visão pode ter nenhuma, uma, ou múltiplas traduções. Uma das dificuldades associada ao problema de tradução de atualização de visão surge quando existem mais de uma tradução possível, e apenas uma deve ser escolhida. Existem algumas formas de ambigüidades que podem ser resolvidas em "tempo de definição" da visão. É o que chamamos de ambigüidades no "nível de esquema", uma vez

que estas são causadas pela existência de mais de um componente do esquema onde se pode realizar a atualização. Nesse caso, a escolha do componente mais apropriado pode ser feita em tempo de definição da visão. No caso das ambigüidades no "nível de dados", que são causadas pela existência de mais de uma instância onde se pode realizar a atualização, não podemos resolvê-las em tempo de projeto. Estas formas de ambigüidades só podem ser resolvidas em tempo de atualização, e para resolvê-las se faz necessário um diálogo com o usuário (em tempo de atualização) para que ele escolha a instância mais apropriada.

Nesta dissertação, apresentamos um enfoque para atualização de bancos de dados objeto-relacionais através de visões XML que utiliza visões de objetos [15] como interface entre a visão XML e o banco de dados. Na nossa proposta, visões XML são atualizadas utilizando uma extensão da linguagem de consulta XQuery definida em[16].

O restante deste Capítulo está organizado como se segue. Na Seção 1.3, apresentamos os trabalhos relacionados e discutimos suas limitações. E, na Seção 1.4, apresentamos o enfoque proposto e os objetivos da dissertação.

1.3 Trabalhos Relacionados

O objetivo principal desta dissertação é a proposta de um mecanismo para atualização de bancos de dados objeto-relacionais através de visões XML. Nesse cenário, duas abordagens principais relacionadas podem ser identificadas: atualização de bancos de dados através de visões de objetos e atualização de bancos de dados através de visões XML. A seguir, essas duas abordagens são discutidas.

1.3.1 Abordagens para atualização de banco de dados através de visões de objetos

O problema de definir traduções para atualizações de visões tem sido considerado por muitos pesquisadores, incluindo [17],[18],[19],[20],[21]. O enfoque de definir procedimentos gerais de tradução [17,18] somente pode ser aplicado a uma classe restrita de atualizações de visões, porque alguns tipos de atualização de visões requerem que mais semântica seja fornecida pelo usuário, para eliminar ambigüidades nas traduções das atualizações em visões [20,21].

O enfoque de Keller propõe que a semântica necessária para remover ambigüidades na tradução das atualizações seja obtida através de diálogo com o usuário na hora da definição da visão. Nesse enfoque as visões são definidas pelo administrador do banco de dados (DBA) que fornece a semântica necessária para escolher um tradutor, a semântica é meramente a seqüência de decisões feitas pelo DBA. As questões são apresentadas para o DBA, sendo fornecida assim uma enumeração completa de todas as traduções possíveis para as atualizações da visão, baseada no esquema da visão, no esquema do banco de dados e nas respostas de questões anteriores. Nesta seção, discutimos os principais enfoques que tratam da atualização de bancos de dados através de visões.

O projeto Penguim [23, 24] propõe um framework unificado para o desenvolvimento de sistemas que compartilham informações. Esse framework armazena os dados em bancos de dados relacionais distribuídos e os apresentam para a aplicação na forma de objetos através das visões de objetos. Visões de objetos projetadas sobre bancos de dados relacionais permitem que cada aplicação tenha seu próprio esquema de objetos ao invés de requerer que todas as aplicações compartilhem o mesmo esquema. Nesse projeto, as relações do banco de dados são mapeadas em templates de objetos através de um diálogo com o usuário, onde cada template pode ser uma combinação complexa de operações de junções e projeções das relações bases.

Para garantir a correta definição dos templates a partir do esquema do banco de dados é usado o modelo estrutural proposto em Penguim [22]. Cada template é ancorado numa relação pivô e a chave desta relação serve como um identificador de objeto quando os objetos são instanciados. Os templates de objetos são arranjados em uma rede de objetos que coletivamente formam o esquema de objetos. Esse processo é guiado pelo conhecimento do usuário, através do uso da semântica da estrutura do banco de dados. Uma vez criada a definição da visão de objetos, o sistema Penguim fornece módulos de instanciação e decomposição de objetos, para suportar as operações de recuperação e armazenamento de objetos. A instanciação é feita a partir dos dados das relações do banco de dados e a decomposição consiste em distribuir as atualizações sobre as relações bases.

O Penguim gera e manipula instâncias de objetos temporários através da ligação das relações base com os templates de objetos pré-definidos. Os usuários atualizam os objetos e finalmente o resultado deve ser colocado à disposição de outros usuários, traduzindo as

atualizações para as tabelas do banco de dados relacional. O módulo de decomposição de objetos faz essa tarefa e mapeia as instancias de objetos de volta para o banco de dados. Esse módulo é chamado quando alguma mudança, feita nas instancias de objetos, deve ser tornadas persistentes no nível do banco de dados. Em tempo de geração do template, todas as possíveis ambigüidades são enumeradas e resolvidas. Quando operações de atualização são executadas nas instâncias, estas atualizações devem ser traduzidas e movidas da representação de objetos para as relações base do banco de dados. Em [22] é apresentado um esquema para tratar operações de atualização em objetos de visão. Atualizações realizadas sobre objetos da visão devem ser traduzidas em operações válidas nas tabelas base. A consistência do banco de dados, definida pelas regras de integridade do modelo estrutural, é mantida para prevenir ou corrigir atualizações ilegais.

1.3.2 Abordagens para atualização de bancos de dados através de visões XML

Na literatura, diversos enfoques para construir visões XML de bancos de dados convencionais têm sido propostos. Muitos deles [5,6,7,8,9] focam o problema de construir visões somente para consultar dados dos bancos de dados. Outros [25,26,27,28,29,30] também provêem a construção de visões XML que atualizam os bancos de dados.

Nesta seção discutimos os principais enfoques que tratam da atualização de bancos de dados através de visões XML.

1.3.2.1 Updating XML

O enfoque Updating XML [28] propõe um conjunto de operações primitivas que modificam a estrutura e conteúdo de um documento XML, tais como: rename, delete, insert e replace. Por conseguinte, é apresentada uma extensão da linguagem de consulta XQuery (XML Query Language), o qual está em processo de padronização na W3C (World Wide Web Consortium). Esta extensão corresponde à implementação destas operações primitivas na sintaxe do XQuery, possibilitando assim operações de atualização ao utilizar a linguagem. No Capítulo 3, apresentamos a sintaxe destas operações de atualização.

O enfoque propõe também algoritmos que possibilitam a atualização de bancos de dados relacional através de atualizações XQuery na visão XML. Além disso, apresenta algumas estratégias de tradução e atualizações XML em atualizações SQL, tais como: Triggers (Per-Tuple e Per-Stratement) e o método Access Support Relations (ASR), o qual é um método para otimizar a validação de expressões de caminho em bancos de dados orientados a objetos e semi-struturados. Por último, o enfoque faz uma análise da performance de diferentes estratégias de atualização.

Entretanto, o enfoque é baseado na premissa que documentos XML são fragmentados (sharedded) e o esquema do banco de dados relacional é derivado de uma DTD. Assim, não discute o caso em que o esquema relacional é predefinido, então, ambos, esquema relacional e definição da visão são entradas do algoritmo.

1.3.2.2 UXQuery

Neste trabalho é proposto um subconjunto da linguagem XQuery, UXQuery [25], para construir visões XML de bancos de dados relacionais que sejam atualizáveis. UXQuery é acrescida de dois novos recursos: a função table, que extrai dados de bancos de dados relacionais e tranforma as tuplas em um conjunto de nós XML, e o operador xnest para facilitar o aninhamento.

Também é definido um método para mapear uma visão XML construída usando UXQuery[26] para um conjunto de visões relacionais correspondentes, as quais podem ser úteis para verificar se a visão XML é atualizável.

Neste enfoque, atualizações nas visões XML são traduzidas para atualizações nas visões relacionais correspondentes. Para traduzir as atualizações nas visões relacionais em atualizações nas tabelas do banco de dados é utilizada a técnica de Dayal e Bernstein [32].

Uma operação de atualização na visão XML é uma tripla < u,Δ, *ref* >, onde u é o tipo da operação (insert, delete, modify); Δ é a árvore XML a ser inserida, ou um valor atômico; e *ref* é uma simples expressão de caminho XPAth a qual indica onde ocorre a atualização. Em remoções não é necessário especificar a Δ, desde que todos os nós filhos de *ref* serão deletados.

Seja o banco de dados da Figura 1.2. A consulta UXQuery da Figura 1.3 (a) gera a visão XML da Figura 1.3 (b). As visões relacionais (Figura1.4) são geradas a partir da visão XML. A seguir, apresentamos alguns exemplos de tradução de atualização na visão XML em atualizações nas visões relacionais.

Author

id	name	email
1	Mary Jones	maryjones@aaa.com
2	Charles Green	charles@bbb.com
3	Michael Kurt	kurt@ccc.com

Conference

confId	confName
DEXA	Conference on Database and Expert Systems Applications
PODS	Symposium on Principles of Database Systems
VLDB	Conference on Very Large Data Bases

Ba

author	isbn
1	1234
1	1235
1	1238
2	1234
2	1237
2	1238
3	1235
3	1236

Paper

pid	title	confId	year
IR	Databases and IR	VLDB	2002
QWEB	Querying the Web	DEXA	2000
WS	Web Survey	VLDB	2001

Pa

author	pid
1	IR
1	QWEB
1	WEB
2	IR
2	WEB
3	WEB

Book

isbn	title	year
1234	Book1	2000
1235	Book2	2001
1236	Book3	2000
1237	Book4	2001
1238	Book5	2001

CONSTRAINTS
On table Paper:
CONSTRAINT ConfPaper
foreign key (confId) references Conference
On table Ba:
CONSTRAINT AuthorBa
foreign key (author) references Author
CONSTRAINT BookBa
foreign key (isbn) references Book
On table Pa:
CONSTRAINT AuthorPa
foreign key (author) references Author
CONSTRAINT PaperPa
foreign key (pid) references Paper

Figura 1.2: Esquema do Banco de Dados

```
<authors>
  {for $a in table("author")
  return
  <author id="{$a/id/text()}">
     {$a/name}
     <address>
        {$a/email}
     </address>
     {xnest $ba in table("ba"), $b in table("b"),
           $pa in table("pa"), $p in table("paper")
      by $year in ($b/year | $p/year)
      where $ba/author=$a/id and $b/isbn=$ba/isbn ar
            $pa/author=$a/id and $p/pid=$pa/pid
      return
      <publications year="{$year/text()}">
         {<book>
            {$b/title}
            {$b/isbn}
         </book>}
         {<conf>
            {$p/title}
            {$p/pid}
         </conf>}
      </publications>
      }
  </author>
  }
</authors>
```

(A

```
<authors>
 <author id="1">
  <name>Mary Jones</name>
   <address>
     <email>maryjones@aaa.com</email>
   </address>
   <publications year="2000">
     <book><title>Book1</title><isbn>1234</isbn></book>
     <conf>
       <title>Querying the Web</title><pid>QWEB</pid>
     </conf>
   </publications>
   <publications year="2001">
     <book><title>Book2</title><isbn>1235</isbn></book>
     <book><title>Book5</title><isbn>1238</isbn></book>
     <conf><title>Web Survey</title><pid>WS</pid></conf>
   </publications>
   <publications year="2002">
     <conf>
       <title>Databases and IR</title><pid>IR</pid>
     </conf>
   </publications>
  </author>
  <author id="2">
   <name>Charles Green</name>
   <address>
     <email>charles@bbb.com</email>
   </address>
   <publications year="2000">
     <book><title>Book1</title><isbn>1234</isbn></book>
   </publications>
   ...
</authors>
```

(B)

Figura 1.3: Consulta UXQuery(A) e Visão XML (B)

```
CREATE VIEW VIEWBOOK AS
SELECT a.id AS id, a.name AS name, a.email AS email, b.year AS year, b.title AS title,
       b.isbn AS isbn
FROM (author AS a LEFT JOIN ba AS ba ON ba.author=a.id) LEFT JOIN book AS b ON b.isbn=ba.isbn

CREATE VIEW VIEWPAPER AS
SELECT a.id AS id, a.name AS name, a.email AS email, p.year AS year, p.title AS title,
       p.pid AS pid
FROM (author AS a LEFT JOIN pa AS pa ON pa.author=a.id) LEFT JOIN paper AS p ON p.pid=pa.pid
```

Figura 1.4: Visões Relacionais

Exemplo 1.1: Inserir um elemento author na visão XML:

U_1: u = *insert*, *ref* = /authors,

```
Δ = {<author id="4">
        <name>Robert White</name>
        <address>
          <email>white@zzz.com</email>
        </address>
      </author>}.
```

Essa inserção é traduzida nas seguintes inserções nas visões relacionais:

```
INSERT INTO VIEWBOOK (id, name, email)
VALUES (4, "Robert White", "white@zzz.com")

INSERT INTO VIEWPAPER (id, name, email)
VALUES (4, "Robert White", "white@zzz.com")
```

Exemplo 1.2: Modifica o elemento title do livro de ISBN=1234 na visão XML:

U_4: u = *modify*, Δ = {Querying the Web using XML},
ref = //book[isbn="1234"]/title.

Essa inserção é traduzida na seguinte inserção na visão relacional VIEWBOOK:

```
UPDATE VIEWBOOK SET title="Querying the Web using XML"
WHERE isbn=1234
```

Exemplo1.3: Inserir um novo author na visão XML:

U_5: u = *delete*, *ref* = //author[@id="1"]
/publications[@year="2000"].

Essa inserção é traduzida nas seguintes inserções nas visões relacionais:

```
DELETE FROM VIEWBOOK
WHERE id=1 AND year=2000

DELETE FROM VIEWPAPER
WHERE id=1 AND year=2000
```

1.3.2.3 SQL Server 2000

O Microsoft SQL *Server* 2000 [29,30] é um SGBD relacional que possui um mecanismo para publicação de seus dados através de visões XML. O ponto chave desse mecanismo é o conceito de esquema anotado. Um esquema anotado consiste em um

documento XML que armazena a estrutura da visão XML juntamente com "anotações" que descrevem o mapeamento entre cada elemento ou atributo da visão XML e o banco de dados. No SQL *Server* 2000, esquemas anotados podem ser definidos usando a linguagem XML *Schema* ou a linguagem XDR (XML *Data Reduced*).

Visões XML definidas no SQL *Server* 2000 podem ser consultadas através de consultas definidas em um subconjunto de *XPath*. Para processar consultas *XPath*, o SQL *Server* 2000 utiliza o esquema anotado para traduzir a *XPath* em uma consulta FOR XML que retorna somente os dados XML requisitados. FOR XML é uma extensão da linguagem SQL para o SQL *Server* 2000 que permite indicar o formato XML do resultado de uma consulta SQL na própria consulta.

No SQL Server, uma visão XML gerada por um XML Schema anotado pode ser modificada usando *updategrams*. Ao invés de usar operações de INSERT, UPDATE ou DELETE, o usuário provê uma imagem anterior (before) da visão XML e uma imagem posterior (after) da visão. O sistema computa a diferença entre essas imagens e gera as operações SQL correspondentes que refletem as mudanças no banco de dados.

1.3.2.4 XSQL Pages Publishing Framework

A tecnologia XSQL *Pages* (páginas XSQL) [31], desenvolvida pela *Oracle* [33], combina o poder de XML, XSLT [34] e XSQL para publicar conteúdo dinâmico na *web* baseado em informações extraídas de bancos de dados objeto-relacionais. Considere, por exemplo, o banco de dados da Figura 1.5 e a página XSQL da Figura 1.6 (V3.xsql), a qual publica o resultado da consulta SQL3 contida em <xsql:query> no formato do esquema XML mostrado na Figura 1.7 (b). O resultado da consulta SQL3 é transformado pelo processador XSQL em um documento XML no formato canônico. A Figura 1.7 (a) mostra o esquema XML do documento XML retornado pelo XSQL para a consulta em V3.xsql. Um exemplo de um documento no formato canônico é mostrado na Figura 1.8. Com o uso da tecnologia XSLT, a estrutura do documento XML canônico pode ser alterada de acordo com o interesse do projetista. Para isso, na página XSQL da Figura 1.6, o atributo href associa um documento stylesheet denominado V3.xsl, que utiliza uma folha de estilo XSLT para transformar o documento XML canônico no formato do esquema XML apresentado na Figura 1.7 (b). A correspondência entre os elementos do esquema XML canônico e os

elementos do esquema XML também são mostradas na Figura 1.7. A Figura 1.8 mostra um exemplo das transformações aplicadas a um resultado da consulta SQL3 na página da Figura 1.6.

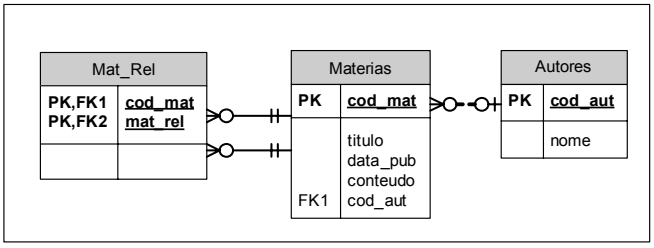


Figura 1.5: Banco de Dados Matérias Relacionadas.

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="V3.xsl"?>
<xsql:query connection="xml" xmlns:xsql="urn:oracle-xsql">
        SELECT m.titulo as titulo,  m.data_pub as data_publicação,  m.conteudo as conteudo,
                ( SELECT a.nome
                  FROM Autores a
                  WHERE m.cod_aut = a.cod_aut) as autor,
                  CURSOR ( SELECT m2.titulo, m2.data_pub
                           FROM Materias m2, Mat_Rel mr
                           WHERE mr.mat_rel = m2.cod_mat
                                 AND mr.cod_mat = m.cod_mat ) as mat_rel
        FROM Materias m
        WHERE m.titulo = {@titulo}
</xsql:query>
```

Figura 1.6: Página XSQL V3.xsql.

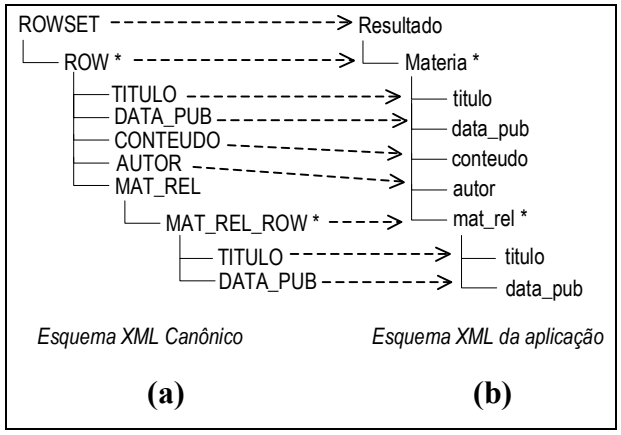


Figura 1.7: Esquemas XML para V3.xsql.

Uma página XSQL é mais versátil quando seus resultados podem mudar com base nos valores de um ou mais parâmetros passados com a requisição. Observe a expressão {@titulo} na cláusula WHERE da consulta SQL3 da página XSQL apresentada na Figura 1.6.

Essa expressão denota um parâmetro de nome titulo. A cada requisição a essa página, a expressão {@titulo} será substituída pelo valor do parâmetro titulo passado na requisição.

A tecnologia XSQL Pages também é utilizada para carga de dados, isto é, inserção, modificação e remoção de dados do banco de dados. Para efetuar uma operação de atualização é necessário saber como está representado o esquema do banco de dados. Assim, XSQL requer um formato canônico para inserções, remoções e atualizações em uma tabela ou visão alvo do banco de dados. Além do mais, XSQL cria uma página para cada operação de atualização.

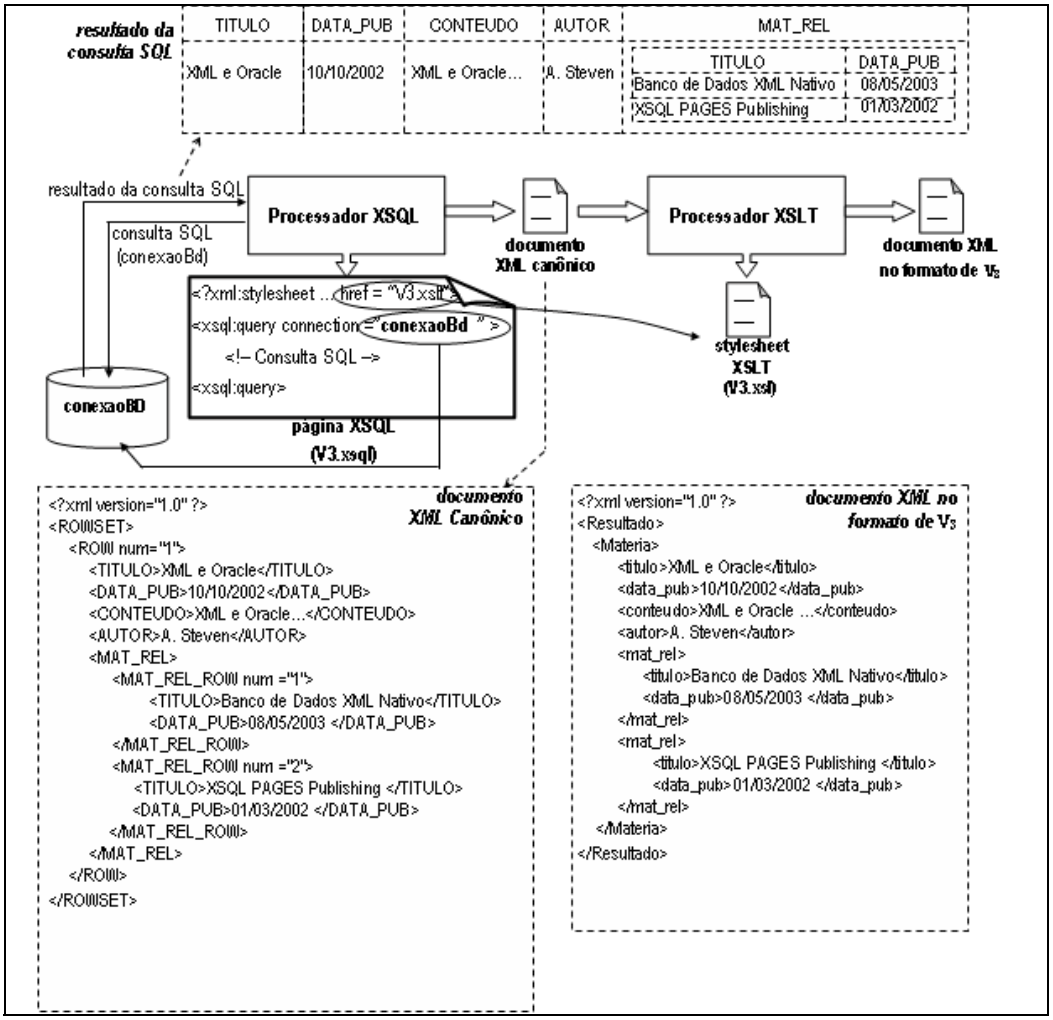


Figura 1.8: Publicação de dados XML baseado em consultas SQL usando XSQL *Pages*.

Na Figura 1.9, apresentamos a página XSQL de uma inserção, a partir de um documento XML, definido numa folha de estilo XSLT (Figura 1.6), de dados na tabela

Autores. Note que, o documento XML da Figura 1.7(B) possui a mesma estrutura da tabela Autores. O resultado dessa inserção é o documento XML:

"<?xml version="1.0" ?> <xsql-status action="xsql:insert-request" rows="1" />".

```
<?xml version="1.0"?>
<xsql:insert-request
connection="conexaoBdteste"
  xmlns:xsql="urn:oracle-xsql"
table="autores"
  transform="InsereAutor.xsl"/>
```

(A)

```
<?xml version="1.0"?>
<ROWSET xsl:version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:for-each select="request/parameters">
    <ROW>
      <COD_AUT>3< COD_AUT >
      <NOME>Wamberg</NOME>
    </ROW>
  </xsl:for-each>
</ROWSET>
```

(B)

Figura 1.9: Página XSQL InsereAutor.xsql (A) e Folha de estilo XSLT InsereAutor.xslt (B)

1.3.2.5 Limitações

Updating XML, UXQuery e SQL Server 2000, apresentam as seguintes limitações: (i) aplicam-se somente a bancos de dados relacionais;(ii) existe uma forte dependência entre a visão XML canônica/Esquema anotado e o banco de dados, de modo que uma mudança no banco de dados requer a redefinição das Visões XML; UXQuery e SQL Server 2000 utilizam linguagens de atualização XML proprietárias; Updating XML não discute o caso em que o esquema relacional é predefinido.

O nosso enfoque se diferencia dos demais, pois utilizamos o modelo objeto-relacional. O problema de atualização de um banco de dados objeto-relacional (BDOR) é bem mais complexo que no caso relacional, pois, além de estruturas relacionais, o modelo objeto-relacional permite o uso de tabelas de objetos, os quais possuem estruturas complexas, como atributos multivalorados (coleções aninhadas), atributos estruturados e de referência. Além do mais, utilizamos XQuery, que é a linguagem de atualização que está em processo de padronização pelo W3C

A tecnologia Oracle XSQL Pages aplica-se ao modelo objeto-relacional, entretanto para efetuar uma operação de atualização é necessário saber como está representado o

esquema do banco de dados. Assim, XSQL requer um formato canônico para inserções, remoções e atualizações em uma tabela ou visão alvo do banco de dados. Além do mais, XSQL cria uma página para cada operação de atualização.

1.4 Enfoque Proposto e Objetivos

Neste trabalho, implementamos o Módulo de Tradução de Atualizações do XML *Publisher*, um *framework* para publicação de dados objeto-relacionais através de visões XML que foi proposto em [35]. No XML *Publisher* os dados são descritos em três níveis de esquema como ilustrado na Figura 1.10. Para publicar uma visão XML, o projetista deve definir uma visão de objeto [15], denominada Visão de Objeto *Default* (VOD), de modo que os objetos da VOD possuem a mesma estrutura dos elementos da visão XML. Visões de objeto provêem uma técnica poderosa para determinar visões lógicas, ricamente estruturadas, sobre um banco de dados já existente. No *Oracle* 9i, uma visão de objeto é definida através de uma consulta SQL3, a qual especifica como os objetos da visão são sintetizados a partir dos dados do banco de dados; e, caso atualizações sejam permitidas, devem ser definidos tradutores (*instead of triggers*) que especificam como atualizações definidas sobre a visão de objeto são traduzidas em atualizações especificadas sobre o banco de dados.

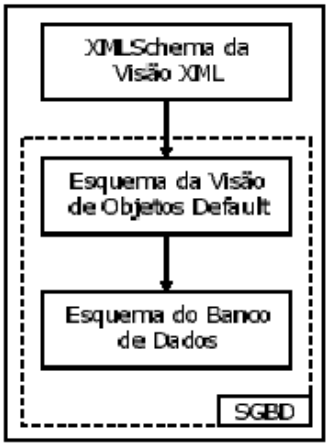
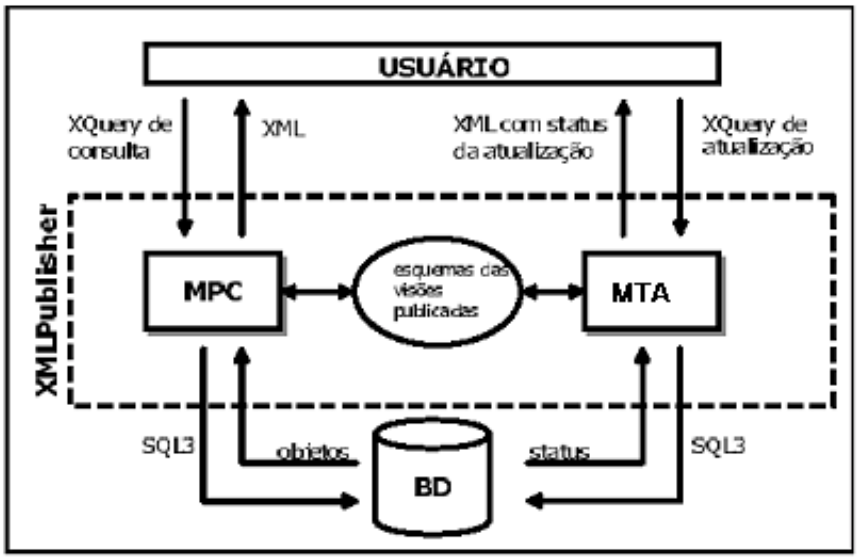


Figura 1.10: Três níves de esquema

No XML *Publisher*, consultas e atualizações definidas sobre o esquema da visão XML são traduzidas em consultas e atualizações sobre o esquema da visão de objeto, e que, por sua vez, são traduzidas pelo mecanismo de visão do SGBD em consultas e atualizações

definidas sobre o esquema do banco de dados. A Figura 1.11 mostra os principais componentes do XML *Publisher*. O Módulo de Processamento de Consultas (MPC) é responsável pelo processamento de consultas *XQuery* definidas sobre a Visão XML e o Módulo de Tradução de Atualização (MTA) pela tradução de atualizações definidas sobre a Visão XML. O MPC foi implementado em [35]. O *XML Publisher* é disponibilizado como uma aplicação web, de forma que seus serviços podem ser acessados por um cliente através de requisições *HTTP GET*.

Figura 1.11: Arquitetura do XML Publisher



Atualizações sobre as visões XML publicadas no XML *Publisher* são definidas através de uma extensão da linguagem *XQuery* [28]. Como apresentado na Figura 1.11, uma atualização submetida ao XML *Publisher* é processada pelo Módulo de Tradução de Atualização (MTA) seguindo os seguintes passos: (i) a atualização *XQuery* é traduzida em uma atualização SQL3 definida sobre a VOD correspondente; (2) as atualizações na visão de objeto *default* são por sua vez traduzidas pelos "*instead of triggers"*, em atualizações definidas sobre o esquema do banco de dados. Um "*instead of trigger"* contém um bloco de comandos PL/SQL [36] para traduzir atualizações aplicadas a uma visão de objetos em atualizações nas tabelas do banco de dados. No Capítulo 5, apresentamos a ferramenta para automatizar o processo de geração dos tradutores de atualizações de visões de objeto.

Nesta dissertação abordamos apenas os aspectos referentes à atualização de dados objeto-relacionais através de visões XML no XML *Publisher*. Os aspectos referentes ao

processamento de consultas *XQuery* e a definição de visões de objetos *default* no XML *Publisher* estão definidos em [35]. Então, baseado no modelo objeto-relacional do SGBD *Oracle* 9i, desenvolvemos os seguintes itens:

1. TAV (*Tradutor de Atualização de Visões*), um protótipo de uma ferramenta semi-automática para criar tradutores ("instead of triggers") de visões de objetos. O processo de criação de um tradutor com TAV tem início com o usuário definindo, através de uma interface gráfica, o tipo dos objetos da visão. O usuário deve, então, carregar o esquema do banco de dados e através de uma GUI, definir as assertivas de correspondência do esquema da visão com o esquema do banco de dados [14,37]. Com base no conjunto de assertivas de correspondência da visão, TAV gera automaticamente os tradutores básicos da visão de objetos.
2. DSG (*Default object view Gerator*), uma ferramenta que gera automaticamente o esquema da visão de objeto d*efault*. O processo de geração da VOD tem início com o usuário definindo, através de uma interface gráfica, o XML *Schema* [38,39] da visão XML. O XML *Schema* [38,39] da visão XML é utilizado para gerar automaticamente o tipo da VOD.
3. Um *software* denominado *XUpdateTranslator*, para a tradução de atualizações *XQuery* no XML *Publisher*. O *XUpdateTranslator* transforma uma atualização *XQuery* numa atualização SQL3 aplicada à respectiva VOD. Em seguida, uma vez que os tradutores ("instead of triggers") da VOD foram criados pelo TAV, o banco de dados é atualizado.

Os capítulos a seguir estão organizados da seguinte forma. No Capítulo 2, apresentamos o Modelo Objeto-Relacional e o mecanismo de Visões de Objetos; no Capítulo 3, apresentamos algumas características da tecnologia XML abordadas neste trabalho; no Capítulo 4, damos uma visão geral do processo de tradução de atualizações no XML *Publisher*; no Capítulo 5, discutimos a geração de tradutores para atualização de bancos de dados objeto-relacionais através de visões de objeto; no Capítulo 6 apresentamos nosso enfoque para atualização de bancos de dados objeto-relacionais através de visões XML; e, no Capítulo 7, apresentamos nossas conclusões e direções para trabalhos futuros.

Capítulo 2 - Modelo Objeto-Relacional

Neste capítulo apresentamos o modelo objeto-relacional baseado no modelode dados do Oracle 9i [40]. O modelo objeto-relacional estende o modelo de dados relacional fornecendo um sistema de tipos mais rico, incluindo orientação a objeto e acrescentando estruturas a linguagens de consulta relacionais, como SQL, para tratar os tipos de dados acrescentados [41]. É importante ressaltar que um esquema objeto-relacional pode ser puramente relacional ou objeto-relacional.

Este capítulo é organizado como se segue. Na Seção 2.1, apresentamos as características do modelo objeto-relacional. Na Seção 2.2, apresentamos uma notação gráfica para representar esquemas objeto-relacionais. Na Seção 2.3, abordamos as características orientadas a objetos da linguagem SQL3 utilizadas nesta dissertação e na Seção 2.4, apresentamos o mecanismo das Visões de Objetos proposto no Oracle 9i.

2.1 Esquema Objeto-Relacional

O modelo objeto-relacional pode ser visto como uma extensão do modelo relacional para suportar a capacidade de modelagem do modelo orientado a objetos. A extensão inclui mecanismos para permitir aos usuários estender o banco de dados com tipos e funções específicas da aplicação.

Nos últimos anos a Oracle Corporation trabalhou na extensão do seu Sistema de Gerência de Banco de Dados Relacional com o objetivo de transformá-lo em um Sistema de Gerência de Banco de Dados Objeto-Relacional adicionando suporte a extensão de tipos, armazenamento de objetos, controle de cache de objetos, extensão de consultas, suporte a tipos de dados multimídia, visões de objetos, entre outras características. O modelo objeto-relacional utilizado neste trabalho é baseado no modelo de dados do Oracle 9i.

No SGBD Oracle 9i [33], um esquema objeto-relacional é uma tripla **S**=(**T**, **R**, **I**), onde **T** é um conjunto de definições de tipos, **R** é um conjunto de tabelas e **I** é um conjunto de restrições de integridade. Uma das principais características do uso de objetos é que além

do usuário poder usar os tipos pré-definidos pelo SGBD pode também construir tipos próprios para sua aplicação. Esses tipos são chamados de Tipos Abstratos de Dados (TAD) que podem ser usados da mesma forma que são usados os tipos pré-definidos.

Um tipo serve como molde para a criação de objetos (instâncias do tipo) e serve para definir a estrutura de dados (atributos) e as operações (métodos) que são comuns às instâncias do tipo. Para criar um TAD em um banco de dados objeto-relacional, usamos o comando CREATE TYPE.

Os atributos de um TAD podem ser classificados em *monovalorados* ou *multivalorados*. O valor de um atributo monovalorado é um objeto (atômico ou estruturado) ou uma referência para um objeto. Já o valor de um atributo multivalorado é uma coleção de objetos ou uma coleção de referências para objetos. Um atributo multivalorado pode ser representado como uma *Nested Table* (coleção ilimitada de objetos) ou *Varray* (coleção limitada de objetos). Considere, por exemplo, o esquema objeto-relacional Pedidos apresentado na Figura 2.1:

```
CREATE TYPE Tcliente AS OBJECT
( codigo INTEGER, nome VARCHAR2(50) );

CREATE TYPE Titem AS OBJECT
( codigo INTEGER, produto VARCHAR2(30),
    quantidade INTEGER );

CREATE TYPE Tendereco AS OBJECT
( rua VARCHAR2(30), cidade VARCHAR2(15),
  estado VARCHAR2(2), cep VARCHAR2(8) );

CREATE TYPE Tlista_item AS TABLE OF Titem ;

CREATE TYPE Tpedido AS OBJECT
( codigo INTEGER, data DATE, dataEntrega DATE,
  enderecoEntrega Tendereco, cliente_ref REF Tcliente,
  listaItens Tlista_item);

CREATE TABLE Clientes OF Tcliente
( codigo PRIMARY KEY );

CREATE TABLE Pedidos OF Tpedido
( codigo PRIMARY KEY, cliente_ref REFERENCES Clientes)
NESTED TABLE listaItens STORE AS listaItens_nt_tab;
```

Figura 2.1: Esquema objeto-relacional Pedidos.

– O atributo nome, do tipo $\mathbf{T}_{\text{cliente}}$, é monovalorado atômico;

- O atributo enderecoEntrega, do tipo T_{pedido}, é monovalorado estruturado cujo valor é uma instância do tipo $T_{endereco}$;
- O atributo cliente_ref, do tipo T_{pedido}, é monovalorado de referência cujo valor é uma referência para uma instância do tipo $T_{cliente}$;
- O atributo listaItens, no tipo $T_{pedido,}$ é multivalorado (Nested Table) de tipo T_{lista_item} cujo valor é uma coleção de instâncias de T_{item}.

O modelo objeto-relacional do Oracle 9i suporta dois tipos de tabela: tabelas de tuplas e tabelas de objetos. As tabelas de tuplas são as tabelas do modelo relacional. Uma tabela de objetos possui associado um tipo e os objetos inseridos na tabela possuem um identificador único (OID), permitindo, assim, que os objetos possam ser referenciados por outros objetos através de atributos de referência. A tabela clientes no esquema Pedidos da Figura 2.1 é uma tabela de objetos, cujos objetos são instâncias do tipo $T_{cliente}$. Estes objetos são referenciados pelas instâncias de T_{pedido} através do atributo cliente_ref. O escopo de um atributo de referência (tabela ou visão que contém os objetos referenciados) é especificado com as restrições de escopo [40]. Por exemplo, na tabela Pedidos, a restrição referencial cliente_ref REFERENCES Clientes especifica que o escopo do atributo de referência cliente_ref é a tabela Clientes.

Antes de inserir um objeto em uma tabela de objetos devemos criá-lo usando um construtor de objetos. Um construtor de objetos é uma função que cria um objeto de um determinado tipo. Considere, por exemplo, o esquema Pedidos apresentado na Figura 2.1. Um exemplo de construtor de objeto para o tipo $T_{cliente}$ é $T_{cliente}$ (1 , "Wamberg"). Um objeto do tipo T_{lista_item} pode ser criado com o construtor T_{lista_item} (T_{item} (10, 'Água', 10), T_{item} (11, 'Suco', 10)). Observe que T_{lista_item} é uma Nested Table que contém duas instâncias do tipo T_{item}.

2.2 Notação Gráfica

A Unified Modeling Language (UML) [42] tem se tornado um padrão para modelagem de objetos durante as fases de análise e projeto do desenvolvimento de softwares. Neste trabalho adotamos uma notação gráfica baseada em [43] que estende o diagrama de classes da UML, através do uso de *estereótipos*, para representar um esquema objeto-relacional. Neste trabalho utilizamos *estereótipos*, que permitem modelar, além de

tipo de objetos (chamado de classe na UML), também tabelas de objetos, tabelas relacionais e visões de objetos.

A Figura 2.2 mostra a representação gráfica do esquema objeto-relacional Pedidos da Figura 2.1. Na notação gráfica, os retângulos com o estereótipo <<Tipo de Objeto>> representam um TAD; os retângulos com o estereótipo <<Tabela de Objeto>> representam tabelas de objetos; os retângulos com o estereótipo <<Tabela Relacional>> representam tabelas de tuplas; e os retângulos com o estereótipo <<Visão de Objeto>> representam uma tabela virtual de objetos. Os atributos cujos tipos são pré-definidos são escritos dentro dos retângulos. Os demais atributos cujos tipos são definidos pelo usuário são representados por setas rotuladas com o nome do atributo. Usamos setas simples para atributos monovalorados e setas duplas para atributos multivalorados. Para os de referência, adicionamos o estereótipo <<REF>> à seta.

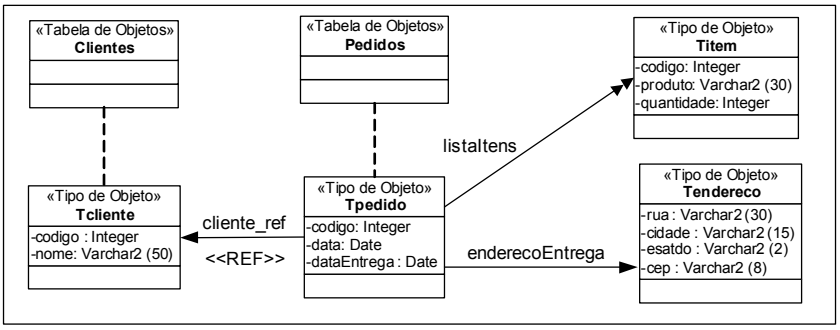


Figura 2.2: Representação gráfica do esquema Pedidos.

2.3 SQL3

SQL (*Structured Query Language*) é a linguagem padrão para a definição e manipulação de dados armazenados em banco de dados relacionais. Ela pode ser considerada uma das maiores razões para o sucesso da tecnologia de bancos de dados relacionais [44]. Em sua mais recente versão (SQL:1999 ou SQL3), adicionou novas características para permitir o suporte a dados orientados a objetos. Alguns SGBD's, como DB2 e Oracle, já dão suporte às características de orientação a objetos da linguagem SQL3.

Algumas das funcionalidades que caracterizam SQL3 como uma linguagem de definição de dados (LDD), orientados a objetos, foram apresentadas na seção anterior (TAD,

tabela de objetos, OID, atributo multivalorado, atributo de referência). A seguir, discutiremos alguns operadores da linguagem SQL3 como uma linguagem de manipulação de dados (LMD) orientados a objeto:

- ***TABLE***

  Esse operador permite desaninhar uma coleção de objetos (*Nested Table* ou *Varray*) dentro de uma cláusula FROM, bem como também é utilizado nas clásulas INSERT, DELETE E UPDATE. Dessa forma, conseguimos acessar individualmente cada objeto de uma coleção. Considere, por exemplo, o esquema Pedidos apresentado na Figura 2.2. Para inserir o item de código 10, produto "livro" e quantidade 5 na coleção de objetos listaitens do pedido de número 1, devemos criar o seguinte comando de atualização SQL3:

```
INSERT INTO TABLE
(SELECT p.listaItens FROM
Pedidos p WHERE p.codigo = 1)

VALUES(10,'livro',5);
```

- ***REF***

  Esse operador pode ser usado em consultas para obter uma referência para um objeto. Considere, por exemplo, o esquema Pedidos apresentado na Figura 2.2. A consulta a seguir retorna o OID de todos os clientes que fizeram algum pedido:

```
SELECT REF(p)
FROM Pedidos p
```

- ***CURSOR***

  Esse operador oferece uma forma de sintetizar coleções de objetos (*Nested Table* ou *Varray*) em consultas. Considere, por exemplo, o esquema relacional Pedidos_rel (Figura 2.6.). Na consulta abaixo, para cada cliente será retornado seu nome e uma coleção de objetos contendo os códigos dos seus pedidos:

```
SELECT c.cnome,CURSOR( SELECT p.pcodigo
                       FROM Pedidos_rel p
                       WHERE c.ccodigo = p.pcliente)
FROM Clientes_rel c
```

2.4 Visões de Objetos

Uma visão relacional é uma tabela virtual que deriva seus dados de outras tabelas. Estas tabelas podem ser tabelas base ou visões previamente definidas. As vantagens de usar visões são: (i) proporcionam um nível adicional de segurança, restringindo o acesso a um conjunto de linhas e colunas de uma tabela, (ii) ajudam a alcançar um certo grau de independência lógica, uma vez que é possível alterar o esquema do banco de dados sem alterar a visão, (iii) permitem o acesso a dados que estão em várias tabelas de forma transparente, assim simplificando o comando para os usuários.

Assim como uma visão relacional é uma tabela virtual, uma visão de objetos é uma tabela virtual de objetos. Além das características das visões tradicionais, as visões de objetos possuem as seguintes características:

- Os objetos das visões possuem identificadores, que fornecem a capacidade para referenciar e serem referenciados por outros objetos;
- Provê a co-existência de aplicações relacionais e OO. Elas tornam mais fácil introduzir aplicações OO para dados relacionais já existentes sem ter que fazer mudanças drásticas de um paradigma para o outro;
- Facilidade para evolução de esquemas. As visões de objetos fornecem a flexibilidade de ver, de mais de uma maneira, o mesmo dado relacional ou orientado a objetos. Sendo assim, cada aplicação pode ter seu próprio esquema orientado a objetos sem ter quer mudar o esquema do banco de dados;
- Consistência e compartilhamento de dados relacionais com novas aplicações baseadas em objetos.

No Oracle 9i, para criar uma visão de objetos, primeiro deve-se criar os tipos da visão. A Figura 2.3 contém a definição dos tipos de uma visão Pedidos_v, cuja representação gráfica é mostrada na Figura 2.4. Depois, então, define-se a visão através de uma consulta SQL3 que especifica como os objetos da visão são sintetizados a partir dos dados do banco de dados. A Figura 2.5 contém a consulta SQL3 que define a visão Pedido_v baseada no esquema do banco de dados relacional Pedido_rel, apresentado na Figura 2.6. A visão

Pedidos_v contém um conjunto de objetos do tipo T_{Pedido}, os quais são sintetizados a partir das tuplas das tabelas base.

```
CREATE TYPE Tcliente_v AS OBJECT
( codigo INTEGER, nome VARCHAR2(50) );

CREATE TYPE Titem_v AS OBJECT
( codigo INTEGER, produto VARCHAR2(30),
    quantidade INTEGER );

CREATE TYPE Tendereco_v AS OBJECT
( rua VARCHAR2(30), cidade VARCHAR2(15),
  estado VARCHAR2(2), cep VARCHAR2(8) );

CREATE TYPE Tlista_item_v AS TABLE OF Titem_v ;

CREATE TYPE Tpedido_v AS OBJECT
( codigo INTEGER, data DATE, dataEntrega DATE,
  enderecoEntrega Tendereco_v, cliente_ref REF
 Tcliente_v,
  listaItens Tlista_item_v);
```

Figura 2.3: Definição dos tipos da visão Pedidos_v.

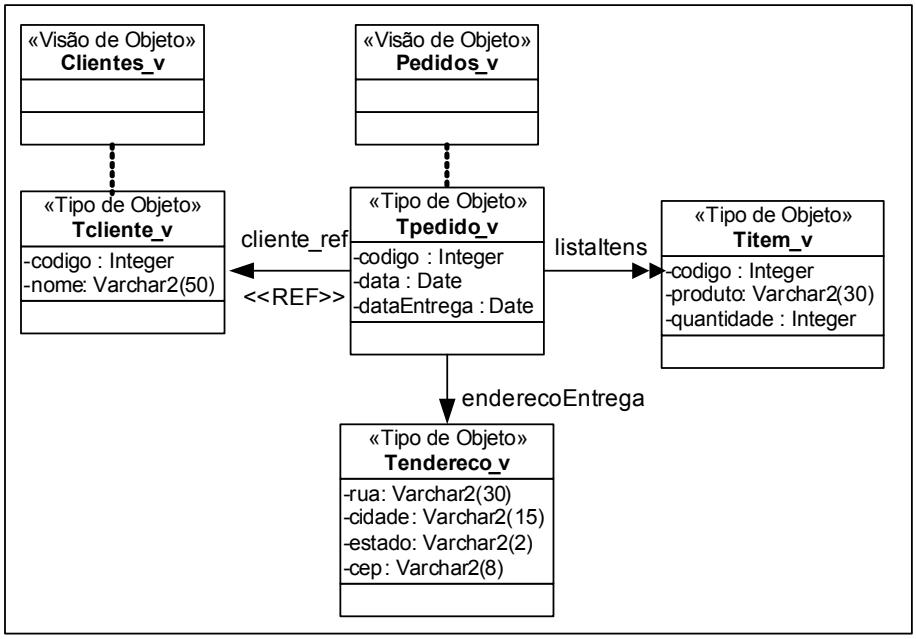


Figura 2.4: Esquema da visão de objetos Pedidos_v.

```
CREATE VIEW Pedidos_v OF Tpedido
WITH OBJECT IDENTIFIER (codigo) AS
   SELECT
Tpedido(
        tpedidos_rel•pcodigo,
        tpedidos_rel•pdata,
        tpedidos_rel•pdataEntrega,
        Tendereco ( tpedidos_rel•prua, tpedidos_rel•pcidade, tpedidos_rel•pestado, tpedidos_rel•pcep ),
        (SELECT REF (tcliente)
         FROM Clientes_v tcliente, Clientes_rel tclientes_rel
         WHERE tpedidos_rel•pcliente = tclientes_rel•ccodigo
               tclientes_rel•ccodigo = tcliente•codigo),
        CAST(MULTISET( SELECT

                              Titem(
                                   titens_rel•icodigo,
                                   (SELECT tprodutos_rel•pnome
                                    FROM Produtos_rel tprodutos_rel
                                    WHERE titens_rel•iproduto= tprodutos_rel•pcodigo),
                                   titens_rel•iquantidade)
                             FROM Itens_rel titens_rel
        ) AS Tlista_item )
FROM Pedidos_rel tpedidos_rel
```

Figura 2.5: Definição da visão Pedidos_v.

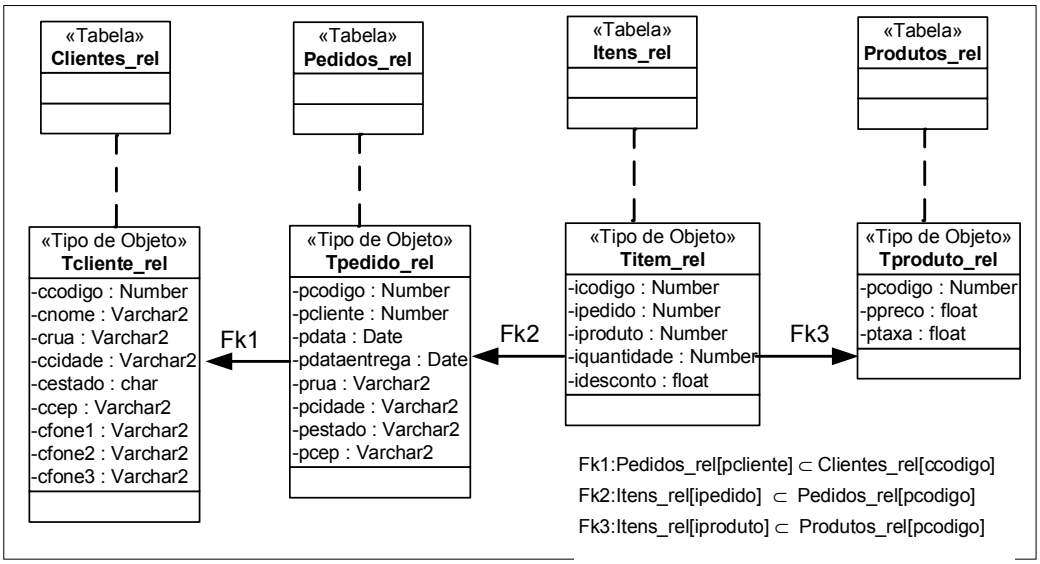


Figura 2.6: Esquema do banco de dados Pedidos_rel.

Consultas SQL3 ,definidas sobre uma visão de objetos, são traduzidas em consultas sobre o banco de dados. No caso em que atualizações sobre uma visão de objetos são permitidas, devemos definir tradutores (“*instead of triggers”*) que especifiquem como

atualizações sobre a visão de objetos são traduzidas em atualizações especificadas sobre o banco de dados. O código do "*instead of trigger"* é executado quando inserções, atualizações ou remoções são solicitadas na visão ou em algum atributo multivalorado da visão. A Figura 2.7 mostra, a título de ilustração, o "*instead of trigger"* Adiciona_Cliente, o qual traduz a adição de um pedido na visão Clientes_v (Figura 2.4) em uma adição na tabela relacional Clientes_rel (Figura 2.6).

```
CREATE TRIGGER Adiciona_Cliente
INSTEAD OF TRIGGER INSERT ON Clientes_v
FOR EACH ROW
   BEGIN
      INSERT INTO CLIENTES_REL
      (ccodigo, cnome)
       VALUES (:new.codigo, :new.nome)
  END
```

Figura 2.7 *Instead of trigger* Adiciona_Cliente.

Capítulo 3 - XML: Modelo de Dados, Linguagens de Esquema e Consulta

A linguagem modelo XML (e**X**tended **M**arkup **L**anguage) [1] vem se tornando um padrão para troca e integração de dados na WEB. Isto cria a necessidade de publicar dados, os quais estão armazenados em bancos de dados convencionais, no formato XML. Neste capítulo apresentamos a linguagem XML. Na seção 3.1, apresentamos uma visão geral sobre XML e a sintaxe básica para a criação de documentos; na Seção 3.2 descrevemos a linguagem padrão de definição de esquemas XML - XML Schema e sua representação gráfica; na Seção 3.3 apresentamos a linguagem de consulta XQuery e descrevemos operadores da sua extensão proposta em [16] para suportar operações de atualização sobre dados XML.

3.1 XML : Sintaxe Básica

XML (*eXtensible Markup Language*) [1] é uma linguagem de marcação desenvolvida pelo W3C (*World Wide Web Consortium*) [45] para descrever informações. Assim como HTML [46], XML tem origem na SGML (*Standard Generalized Markup Language*), que é um padrão internacional para definição de formatos de representação de texto em meio eletrônico. Entretanto, ao contrário de HTML, que foi projetada para descrever a apresentação dos dados, XML foi projetada para descrever o conteúdo dos dados. A seguir, destacamos alguns aspectos que tornam a linguagem XML mais poderosa que HTML:

- **Independência de conteúdo e apresentação**. XML aborda apenas o conteúdo de um documento. A apresentação pode ser tratada por linguagens específicas para apresentação, tais como: *Cascading Style Sheets* (CSS) e *eXtensible Stylesheet Language* (XSL);
- **Linguagem Extensível**. XML permite que os usuários definam novas marcas de acordo com o domínio que está sendo modelado. Estas marcas servem para descrever o conteúdo de um documento;

- **Validação.** Um documento XML pode ser associado a uma definição de esquema XML, a qual define a estrutura do documento. Assim, aplicações podem validar seus dados de acordo com um esquema;
- **Linguagem flexível**. XML permite a apresentação de um mesmo conteúdo em diversos formatos.

XML também pode ser vista como uma metalinguagem, pois, por ser extensível, permite a criação de outras linguagens de marcadores. Estas linguagens podem ser definidas para domínios específicos (matemática, química, comércio eletrônico, entre outros). Assim, os marcadores podem capturar mais semântica sobre os dados que estão sendo modelados, o que não ocorre em um documento HTML, pois seus marcadores têm função apenas de formatar o texto para apresentação.

Um documento XML consiste de vários elementos. ***Os elementos são os blocos principais da estreutura hierárquica de um documento XML.*** Um elemento é descrito por um marcador inicial (i.e, <nome_do_elemento>) e um marcador final (i.e, </nome_do_elemento>). que delimitam o conteúdo do elemento. Cada elemento pode conter texto, conter outros elementos aninhados (subelementos), ter conteúdo misto (texto e subelementos) ou ser vazio.

Exemplo 3.1: Considere o documento XML abaixo:

```
<pessoa>
  <nome>Wamberg</nome>
  <e-mail>wamberg@lia.ufc.br </e-mail>
</pessoa>
```

Os marcadores <pessoa> e </pessoa> descrevem a estrutura do elemento pessoa. O elemento pessoa é composto por dois sub-elementos: nome e e-mail. Como podemos observar, os marcadores de um documento são balanceadas, ou seja, são fechadas na ordem inversa da que foram abertas.

Um elemento pode possuir um ou mais atributos. Estes são especificados no marcador inicial do elemento. O valor de um atributo é sempre um texto e deve aparecer entre aspas.

Exemplo 3.2: Considere o documento XML apresentado no Exemplo 3.1 acrescido do CPF como atributo do elemento pessoa. O novo documento é apresentado a seguir:

```
<pessoa cpf="75700998315">
  <nome>Wamberg </nome>
  <e-mail>wamberg@lia.ufc.br </e-mail>
</pessoa>
```

Um documento XML pode ser considerado bem formado e válido. Para ser bem formado, o documento deve obedecer algumas regras, tais como: (1) conter pelo menos um elemento; (2) conter um único marcador inicial e marcador final descrevendo um elemento que contenha todo o documento (chamado elemento raiz); e (3) todas as outras marcas devem estar aninhadas apropriadamente. Os documentos XML apresentados anteriormente (Exemplo 3.1 e Exemplo 3.2) são documentos XML bem formados.

Para ser válido um documento precisa ser bem-formado e deve estar de acordo com a gramática que define sua estrutura. Esta gramática é definida através de um esquema XML, que descreve a estrutura de um documento, determinando os elementos que podem participar deste documento e também os que podem estar associados a esses elementos. A Seção 3.2 mostra como definir esquemas para documentos XML.

A verificação sintática de um documento XML, que identifica se este é ou não bem formado, é realizada por um *parser*. Os *parsers* de validação, além de detectar se um documento é bem formado, são capazes de verificar se o mesmo está de acordo com o seu esquema XML associado, ou seja, verifica se o documento é válido ou não.

3.2 Esquemas XML

Um esquema descreve a estrutura lógica de uma fonte de informação, incluindo os tipos dos dados, os relacionamentos entre os dados e as restrições envolvendo os dados. Para XML, um esquema é usado para descrever os elementos, o conteúdo dos elementos, a lista de atributos de um determinado elemento e as restrições sobre os elementos/atributos. Além disso, um esquema é útil para validar o conteúdo de um documento, ou seja, para determinar se um documento está de acordo com a gramática definida pelo esquema. E essa gramática pode ser reutilizada para validar e definir a estrutura de outros documentos.

A funcionalidade de validar um documento XML é muito importante no contexto de aplicações *web* que trocam informações entre diversas fontes, pois com a definição de esquema é possível verificar se um determinado documento está de acordo com a estrutura esperada, facilitando o processamento de dados pelas aplicações.

A primeira linguagem proposta para definição de esquemas XML foi a DTD (*Document Type Definition*) [47]. Entretanto, a DTD possui algumas limitações como pode ser observado em [48]. Para sobrepor as limitações da DTD, outras linguagens foram propostas. Entre elas, destacamos: XML Schema [38,39] e RDF [49]. Estas linguagens são mais ricas em semântica e oferecem recursos adicionais para definição de esquemas. Uma análise comparativa dessas e de outras linguagens para definição de esquemas XML é apresentada no trabalho [50].

Os esquemas das visões XML publicadas pelo XML *Publisher* são definidos usando um subconjunto da linguagem XML *Schema*. Sendo assim, na seção a seguir discutimos as características dessa linguagem, destacando apenas as características implementadas no XML *Publisher*.

3.2.1 XML Schema

A linguagem XML *Schema* introduz novos construtores que a tornam mais expressiva que DTD e permite não só especificar a sintaxe de um documento XML, mas também: especificar o tipo de dados do conteúdo de cada elemento; herdar sintaxe de outros esquemas; criar tipos de dados simples e complexos; especificar o número mínimo e máximo de vezes que um elemento pode ocorrer; entre outros. Com isso, XML *Schema* pode ser utilizada em uma maior variedade de aplicações, além de ter a vantagem de ser escrita em sintaxe XML, não sendo necessário, portanto, aprender uma nova sintaxe. Essa facilidade nos permite utilizar qualquer ferramenta que trabalha com documentos XML para trabalhar com XML *Schema* [51].

Na Figura 3.2, apresentamos um exemplo de uso da linguagem XML *Schema*. Como pode ser observado, trata-se de um documento XML cujo elemento raiz é *Schema*. Na linguagem, todas as declarações de elementos, atributos e tipos devem ser inseridas entre os marcadores do elemento *Schema*.

```
<bib>
 <livro autoresref1="A">
   <ano> 1999 </ano>
   <isbn> 3826561422 </isbn>
   <titulo> Transaction Management in Multidatabase Systems
</titulo>
   <editora> Shaker-Verlag </editora>
 </livro>
 <artigo autoresref="A1 A2">
   <ano> 2000 </ano>
   <cdu> 681.31:061.68 </cdu>
   <titulo> Temporal Serialization Graph Testing </titulo>
   <local_ publicacao> XV SBBD </local_ publicacao>
 </artigo>
 <autor id_autor1="A" instituicaoref="I">
   <nome> Ângelo Brayner </nome>
   <e-mail> brayner@unifor.br </e-mail>
 </autor>
 <autor id_autor="A2" instituicaoref="I1 I2">
   <nome> José Maria Monteiro </nome>
   <e-mail> zemaria@lia.ufc.br </e-mail>
  </autor>
<instituicao id_instituicao="I">
  <nome> Unifor </nome>
  <endereco>
    <cidade> Fortaleza </cidade>
    <estado> Ceará </estado>
    <pai> Brasil </pais>
  </endereco>
</instituicao>
<instituicao id_instituicao="I2">
  <nome> UFC </nome>
  <telefone> 288-9845 </telefone>
  <endereco>
    <cidade> Fortaleza </cidade>
    <estado> Ceará </estado>
    <pais> Brasil </pais>
  </endereco>
</instituicao
</bib>
```

Figura 3.1: ***Bib.xml***

A seguir, descrevemos os principais componentes da linguagem XML *Schema*. Para exemplificar o uso destes componentes, utilizamos o esquema que representa a estrutura do documento Bib.xml, apresentado na Figura 3.1.

Declaração de Elemento

Elementos são declarados utilizando o marcador <element>. Os principais atributos deste são: name, que associa o elemento declarado a um nome; type, que atribtui um tipo ao elemento; e minOccurs e maxOccurs, que especificam a restrição mínima e máxima de ocorrência do elemento, respectivamente: se a restrição mínima for igual a zero, o elemento é opcional, caso contrário é obrigatório; e se a restrição máxima for igual a um, o elemento é mono-ocorrência, caso contrário e multi-ocorrência.

Considere, por exemplo, o elemento autor. A declaração <element name="autor" type="Tautor" minOccurs="0" maxOccurs="unbounded"/> especifica que este elemento é do tipo Tautor e a restrição de ocorrência é opcional e multi-ocorrência.

Declaração de Atributo

Atributos são declarados utilizando o marcador <attribute>. Os principais atributos deste são: name, que associa o atributo declarado a um nome; type, que atribui um tipo ao atributo (o tipo do atributo indica a sua cardinalidade: se o tipo do atributo for IDREFS, este é multivalorado, caso contrário é monovalorado) e use, que especifica a restrição de

ocorrência do atributo: se o use não estiver explícito na declaração do atributo ou se seu valor for *optional*, o valor do atributo é opcional, caso contrário é obrigatório.

Considere o atributo id_instituição. A declaração <attribute name="id_instituição" type="ID" use="required"/> especifica que este atributo é do tipo ID é monovalorado e obrigatório.

```
1 <Schema>
2  <element name="bib">
3   <complexType>
4    <sequence>
5       <element name="livro" type="Tlivro" minOccurs="0"
maxOccurs="unbounded"/>
6    <element name="artigo" type="Tartigo" minOccurs="0"
maxOccurs="unbounded"/>
7     <element name="autor" type="Tautor" minOccurs="0"
maxOccurs="unbounded"/>
8    <element name="instituicao" type="Tinstituicao"
minOccurs="0" maxOccurs="unbounded"/>
9   </sequence>
10   </complexType>
11 </element>

12  <complexType name=" Tpublicacao">
13  <sequence>
14 <element name="ano" type="String"/>
15 <choice>
16   <element name="isbn" type="Integer"/>
17   <element name="cdu" type="Integer"/>
18   </choice>
19   <element name="titulo" type="String"/>
20  </sequence>
21        <attribute name="autoresref" type="IDREFS"
use="required"/>
22 </complexType>

23 <complexType name="Tartigo">
24  <complexContent>
25   <extension base="Tpublicacao" >
26    <sequence>
27     <element name="local-publicacao" type="String"/>
28    </sequence>
29   </extension>
30  </complexContent>
31 </complexType>
32 <complexType name="Tlivro">
33  <complexContent>
34   <extension base="Tpublicacao">
35    <sequence>
36     <element name="editora" type="String"/>
37    </sequence>
38   </extension>
39  </complexContent>
40 </complexType>

41 <complexType name=" Tautor">
42  <sequence>
43   <element name="nome" type="String"/>
44   <element name="e-mail" type="String"/>
45  </sequence>
46  <attribute name="id_autor" type="ID" use="required"/>
47        <attribute name="instituicaoref" type="IDREF"
use="optional"/>
48 </complexType>

49 <complexType name=" Tinstituicao">
50  <sequence>
51   <element name="nome" type="String"/>
52       <element name="telefone" minOccurs="0"
maxOccurs="" />
53   <element name="endereco" type="Tendereco"/>
54  </sequence>
55        <attribute name="id_instituicacao" type="ID"
use="required"/>
56 </complexType>

57 <complexType name="Tendereco">
58  <sequence>
59   <element name="cidade" type="String"/>
60   <element name="estado" type="String"/>
61   <element name="pais" type="String"/>
62  </sequence>
63 </complextype>

64 </Schema>
```

Figura 3.2: XML Schema de ***Bib.xml***

Definição de Tipos

Os tipos restringem o conteúdo permitido para um elemento ou atributo. Existem duas espécies de tipos em XML *Schema*: tipos simples (simpleType) e tipos complexos (complexType). Um tipo simples restringe o conteúdo de um atributo ou o conteúdo textual de um elemento. Um tipo complexo restringe o conteúdo permitido para elementos, em termos dos seus atributos ou sub-elementos.

Na linguagem XML *Schema*, para definir um tipo complexo, usamos o marcador <complexType>. Os tipos simples não precisam ser definidos, pois a especificação da XML *Schema* contempla um conjunto expressivo de tipos primitivos [39]. No entanto, é possível criar novos tipos de dados simples que correspondem a refinamentos de tipos primitivos. Para esse propósito, usamos o marcador <simpleType>.

O atributo name dos marcadores <simpleType> e <complexType> define o nome do tipo que está sendo definido. Quando esses marcadores não possuem o atributo name, dizemos que se trata de uma declaração de tipo anônima. Neste caso, a definição do tipo vem aninhada na declaração de um elemento. Por exemplo, no esquema da Figura 3.2, a declaração do elemento bib (linhas 2 a 11) contém uma definição de tipo anônima, que especifica que o elemento bib contém os elementos artigo, livro, autor e instituicao.

Nas declarações de tipos complexos, além das restrições de ocorrência declaradas para elementos individuais, a XML *Schema* provê restrições a serem aplicadas a grupos de elementos. Essas restrições também podem ser chamadas de delimitadores de grupos. Existem três tipos de delimitador de grupo:

- *sequence*, indica que os elementos do grupo devem aparecer no documento na mesma ordem em que foram declarados no esquema. Por exemplo, na Figura 3.2 (linhas 49 a 56), a declaração do tipo complexo Tinstituicao especifica que os elementos nome, telefone e endereco, contidos neste tipo, devem aparecer nesta ordem no documento. Os tipos complexos definidos nos esquemas XML das visões XML publicadas no XML *Publisher*, usam apenas esse delimitador de grupo;
- *all*, estabelece que todos os elementos do grupo podem aparecer uma ou nenhuma vez, e que eles podem aparecer em qualquer ordem;
- *choice*, indica que somente um dos elementos declarados no grupo pode aparecer no documento.

Um tipo complexo também pode ser definido a partir da extensão de um outro tipo complexo. No esquema da Figura 3.2 a definição do tipo Tartigo (linhas 23 a 31) especifica que o tipo complexo Tartigo é definido a partir do tipo complexo Tpublicacao (linha 25). Assim, um elemento do tipo complexo Tartigo contém os elementos do tipo complexo Tpublicacao, além do elemento local-publicacao. Neste caso, podemos dizer que Tartigo é um subtipo de Tpublicacao.

Restrições de Integridade

XML Schema oferece, além do ID/IDREF/IDREFS, outros mecanismos de restrição de integridade, tais como: key e keyref. Estas restrições de integridade oferecem vantagens em relação ao ID/IDREF/IDREFS, tais como: (i) podem ser aplicadas tanto a elementos como a atributos; (ii) podem ser aplicadas a mais de um elemento ou atributo; (iii) permitem limitar o escopo, no qual o valor do elemento ou do atributo deve ser único, ou deve referenciar; e (iv) a restrição keyref permite capturar o tipo dos elementos ou dos atributos que estão sendo referenciados através do escopo da chave referenciada.

Para exemplificar a especificação de restrições de integridade na linguagemXML Schema, redefinimos o esquema XML Schema Bib_1. Esta nova versão é apresentada na Figura 3.3.

Restrição de Integridade de Chave

XML Schema permite especificar, através do construtor key, que o valor de um conjunto de elementos ou atributos deve ser único dentro de um determinado escopo e, além de único, pode ser referenciado.

Considere a restrição de chave key#3 (linhas 12 a 16). Esta restrição especifica que os valores dos elementos nome e e-mail são únicos dentro do escopo do conjunto de elementos $autor_1$ selecionados pela expressão de caminho $bib_1/autor.

A Figura 3.4 mostra como uma key é declarada. Como podemos observar, o construtor key contém:

- um atributo name (N), que identifica unicamente a restrição.
- um elemento selector (δ_1), que especifica o escopo, através de uma expressão de caminho, para o qual a restrição é declarada.

• um ou mais elementos field (δ_{11},..., δ_{1n}), que determina(m), através de uma ou mais expressões de caminho, o conjunto de elementos ou atributos que deve ser único dentro do escopo especificado pelo selector.

```
1 <schema>
2  <element name=”bib”>
3   <key name=”key#1”>
4    <selector xpath=”artigo”/>
5    <field xpath="isbn"/>
6    <field xpath="título"/>
7   </key>
8   <key name=”key#2”>
9    <selector xpath=”livro”/>
10    <field xpath="isbn"/>
11   </key>
12   <key name="key#3">
13    <selector xpath="autor"/>
14    <field xpath="nome"/>
15    <field xpath="e-mail"/>
16   </key>
17   <key name=”key#4”>
18    <selector xpath=”instituição”/>
19    <field xpath="@id_instituição"/>
20   </key>
21   <keyref name="keyref#1" refer=”key#3”>
22    <selector xpath="artigo"/>
23    <field xpath="nome-autor"/>
24    <field xpath="e-mail-autor"/>
25   </keyref>
26   <keyref name="keyref#2" refer=”key#3”>
27    <selector xpath="livro"/>
28    <field xpath="nome-autor"/>
29    <field xpath="e-mail-autor"/>
30   </keyref>
31   <keyref name="keyref#3" refer=”key#4”>
32    <selector xpath="autor"/>
33    <field xpath="@instituiçãoref"/>
34   </keyref>

35   <complexType>
36    <sequence>
37      <element name=”artigo” type=”Tartigo”
  minOccurs=”0“ maxOccurs=”unbounded”/>
38      <element name=”livro” type=”Tlivro”
 minOccurs=”0“ maxOccurs=”unbounded”/>
39      <element name=”autor” type=”Tautor”
 minOccurs=”0”  maxOccurs=”unbounded”/>
40      <element name=”instituição” type=”Tinstituição”
 minOccurs=”0”  maxOccurs=”unbounded”/>
41    </sequence>
42   </complexType>
43  </element>

44  <complexType name=” Tpublicação”>
45   <sequence>
46    <element name=”ano” type=”Integer”/>
47    <element name=”isbn” type=”String”/>
48    <element name=”título” type=”String”/>
49    <element name=”nome-autor” type="String"
minOccurs=”1” maxOccurs=”unbounded”/>
50    <element name=”e-mail-autor” type="String"
minOccurs=”1” maxOccurs=”unbounded”/>
51   </sequence>
52  </complexType>

53  <complexType name=”Tartigo”>
54   <complexContent>
55    <extension base=”Tpublicação” >
56     <sequence>
57      <element name=”local-publicação” type=”String”/>
58     </sequence>
59    </extension>
60   </complexContent>
61  </complexType>

62  <complexType name=”Tlivro”>
63   <complexContent>
64    <extension base=”Tpublicação”>
65     <sequence>
66      <element name=”editora” type=”String”/>
67     </sequence>
68    </extension>
69   </complexContent>
70  </complexType>

71  <complexType name=”Tautor”>
72   <sequence>
73    <element name=”nome” type=”String”/>
74    <element name=”e-mail” type=”String”/>
75   </sequence>
76   <atribute name=”instituiçãoref1” type=”IDREF”/>
77  </complexType>

78  <complexType name=”Tinstituição”>
79   <sequence>
80    <element name=”nome” type=”String”/>
81    <element name=”endereço” type=”String”/>
82   </sequence>
83   <atribute name=”id_instituição” type=”ID”/>
84  </complexType>

85 </schema>
```

Figura 3.3: ***de integridade***

```
<key name="N">
  <selector xpath="δ1">
  <field xpath="δ11">
        ...
  <field xpath="δ1n">
</key>
```

Figura 3.4: Declaração de Restrição de Chave

Restrição de Integridade Referencial

XML Schema permite declarar, através do construtor keyref, que o valor de um conjunto de elementos ou atributos é uma referência ao valor de um outro conjunto de elementos ou atributos.

Considere a restrição referencial keyref#1 (linhas 21 a 25). Esta restrição especifica que, dentro do escopo do conjunto de elementos em $bib/artigo, os elementos nome-autor e e-mail-autor destes elementos referenciam a chave key#3. Ou seja, para qualquer $p em bib_1$/artigo, existe um $a em bib_1$/autor tal que $p/nome-autor = $a/nome e $p/e-mail-autor = $a/e-mail.

A Figura 3.5 mostra como uma keyref é declarada. O construtor keyref contém:

- um atributo name (ref), que identifica unicamente a restrição.
- um atributo refer (N), que indica a restrição de chave associada à referência.
- um elemento selector (δ_2), que especifica o escopo, através de uma expressão de caminho, para o qual a restrição é declarada.
- um ou mais elementos field (δ_{21}, ..., δ_{2n}), que determina(m), através de uma ou mais expressões de caminho, o conjunto de elementos ou atributos que deve corresponder ao conjunto de elementos ou atributos especificado pela restrição de chave correspondente.

```
<keyref name="ref" refer="N">
  <selector xpath="δ2">
  <field xpath="δ21">
        ...
  <field xpath="δ2n">
</keyref>
```

Figura 3.5: Declaração de Restrição de Integridade Referencial

3.3.2 Representação Gráfica para XML Schema

Nesta seção apresentamos a notação utilizada neste trabalho para representar graficamente a estrutura definida por um XML *Schema*. Esta notação utiliza uma representação estruturada em "árvore de diretórios", onde cada nó da árvore representa um elemento ou atributo declarado no respectivo esquema XML.

Para cada declaração de elemento ou atributo é gerado um nó na árvore. Se o elemento for de um tipo complexo, será gerado um nó-filho na árvore para cada item da sua estrutura. Cada nó deve ser rotulado com o nome do elemento ou atributo, seu tipo (se não for anônimo) e sua restrição de ocorrência. Para atributos, seu nome é precedido do símbolo "@". Se o elemento ou atributo é monocorrência e obrigatório, a restrição de ocorrência é vazia. Se o elemento for monocorrência e opcional, a restrição de ocorrência é "?". Se o elemento for multiocorrência e opcional, a restrição de ocorrência é "*". Finalmente, se o elemento for multiocorrência e obrigatório, a restrição de ocorrência é "+".

A árvore mostrada na Figura 3.6 acima é a representação gráfica do esquema XML apresentado na Figura 3.2. O primeiro nó da árvore é bib, e representa o elemento raiz bib declarado na linha 2 do XML *Schema*. Como podemos perceber no XML *Schema*, a declaração de tipo desse elemento é anônima. Sendo assim, sua representação gráfica não acompanha um tipo como nos demais nós da árvore. Os nós filhos de bib são livro, autor e instituicao, pois no XML *Schema* esses elementos estão declarados como o conteúdo do tipo complexo de bib (linhas 3 a 10 da Figura 3.2). Cada um desses nós filhos são rotulados com o nome do respectivo elemento, o símbolo "*", que denota a restrição de ocorrência zero ou muitos, e o nome do respectivo tipo.

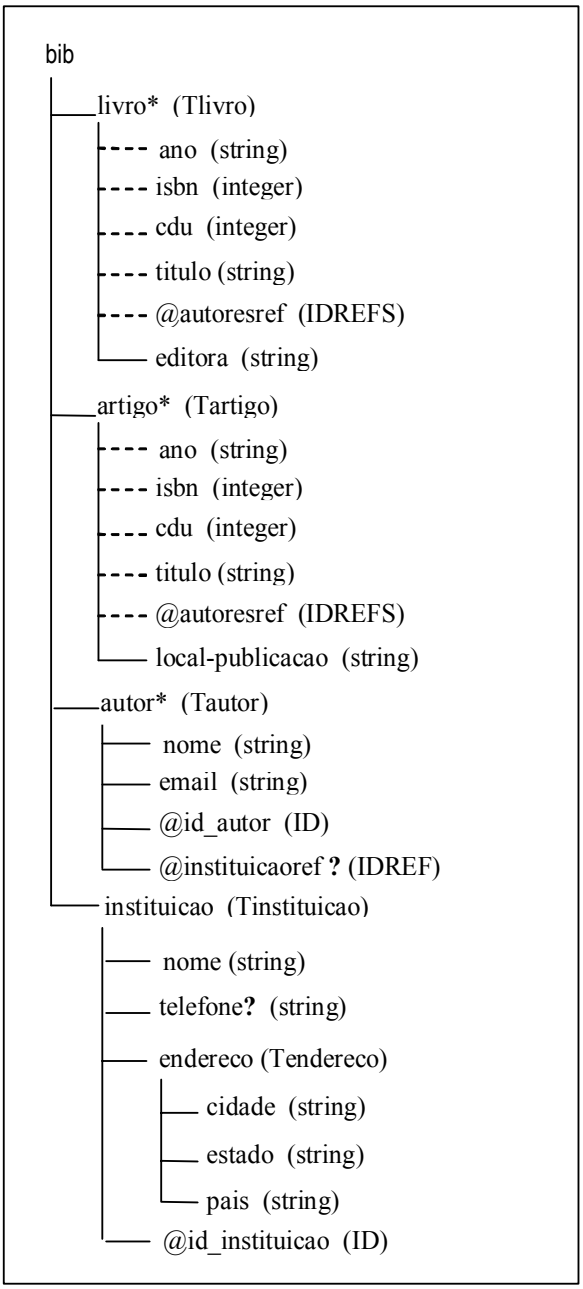


Figura 3.6: Representação gráfica para Bib.xml.

3.3 Linguagens de Consulta para XML

Como grandes quantidades de dados estão sendo armazenadas, trocadas e publicadas utilizando o modelo XML. A habilidade de consultar inteligentemente bancos de dados XML tornou-se muito importante. Entretanto, dados XML são diferentes dos dados de banco de dados relacionais ou orientados a objetos. Em bancos de dados relacionais e orientados a objetos, existe a noção de esquema fixo, pré-definido e independente dos dados.

Em XML, o esquema existe juntamente com os dados, bem como, dados XML não seguem uma estrutura rígida. Assim, linguagens de consulta convencionais como SQL e OQL não são adequadas para especificar consultas em documentos XML. Devido a essa flexibilidade da linguagem XML, devemos ter uma linguagem de consulta XML que possua características para retornar e interpretar as informações consultadas.

Muitas linguagens de consulta para XML têm sido propostas, como, XML-QL [2], XSL [34], XQL [52], *XPath* [3] e *XQuery* [4]. Neste trabalho, utilizamos a linguagem de consulta XQuery, a qual está em processo de recomendação pela W3C e agrega características de diversas outras linguagens de consulta para XML [52,53], bem como SQL e OQL.

As visões XML publicadas no XML *Publisher* podem ser atualizadas usando a extensão da linguagem *XQuery* que será apresentada na Seção 3.3.3 e no Capítulo 6. *XQuery* usa *XPath* para endereçar partes do documento XML consultado. Sendo assim, descrevemos a seguir os principais aspectos das linguagens *XPath* e *XQuery* propostas pelo W3C.

3.3.1 *XPath*

A linguagem *XPath* é uma recomendação do W3C para acessar partes de um documento XML. Esta linguagem usa uma notação de caminhos para navegar através da estrutura hierárquica de um documento XML e foi projetada para ser inserida em outras linguagens chamadas de *host languages*, tais como XSLT e *XQuery*.

XPath modela um documento XML como uma árvore de nós. Este modelo em árvore é apenas conceitual e possui diferentes tipos de nós, tais como, nós-documento, nós-elemento, nós-texto, nós-atributo, entre outros. Para cada tipo de nó, há um modo de determinar seu valor literal (*string-value*). Para alguns tipos de nós o valor é parte do nó; para outros, esse valor é computado a partir do valor de nós descendentes. Por exemplo, o valor de um nó-elemento consiste da concatenação do valor de todos os nós-texto descendentes do nó-elemento na ordem do documento.

O nó-documento encapsula todo o documento XML; ele é o ponto de partida na árvore que descreve o documento XML. Os nós-elemento encapsulam os elementos XML, e podem ter nós-elemento, nós-texto e outros como seus filhos. Nós-elementos também podem conter nós- atributos.

A linguagem XPath é utilizada para navegar através de caminhos em um documento XML. Está sendo desenvolvida pelos grupos *W3C XML Query Working Group* e *XSL Working Group* e projetada para ser inserida em outras linguagens (chamadas de *host language*), como por exemplo, XQuery.

XPath utiliza o modelo de dados de consulta XML do W3C, o qual representa um documento XML como uma árvore rotulada nos nós. XPath explora o contexto hierárquico e seqüencial de elementos em um documento XML para localizar um elemento de interesse.

O bloco de construção básico da linguagem XPath é a expressão. A linguagem provê diversos tipos de expressão que podem ser construídos a partir de palavras-chave, símbolos e operandos. Em geral, os operandos de uma expressão são outras expressões. XPath é uma linguagem funcional que permite que vários tipos de expressão sejam aninhados.

A expressão que localiza um nó em uma árvore e retorna uma seqüência de nós distintos na ordem do documento é chamada de expressão de caminho. Tal expressão consiste de um ou mais passos. Cada passo representa um movimento ao longo do documento em uma determinada direção e retorna uma lista de nós que servem como um ponto de partida para o próximo passo.

A seguir, exemplificamos algumas expressões de caminho utilizadas para consultar o documento XML bib.xml apresentado na Figura 3.1.

***Exemplo 3.3*:** A expressão de caminho **document**("bib.xml")/**bib**/**artigo**/**titulo** consiste em quatro passos: o primeiro passo seleciona o documento XML bib.xml; o segundo seleciona o elemento raiz de bib.xml; o terceiro seleciona os elementos artigo, filhos do elemento raiz e o quarto passo seleciona os elementos titulo dos elementos artigo recuperados no passo anterior. Assim, essa expressão retorna todos os elementos titulo contidos nos elementos artigo, contidos no elemento bib do documento bib.xml. O exemplo abaixo mostra o XML retornado por essa consulta:

```
<titulo> Temporal Serialization Graph Testing </titulo>
```

Para simplificar o próximo exemplo, utilizamos a variável **$bib** para substituir a expressão **document**("bib.xml")/**bib**:

***Exemplo 3.4*:** A expressão de caminho **$bib/livro [ano = "1999"]** retorna todos os elementos livro em **$bib** que possuem o elemento ano igual a "1999". O exemplo abaixo mostra o XML retornado por essa consulta:

```
<livro autoresref="A">
  <ano> 1999 </ano>
  <isbn> 3826561422 </isbn>
  <titulo> Transaction Management in Multidatabase Systems </titulo>
  <editora> Shaker-Verlag </editora>
</livro>
```

3.3.2 XQuery

XQuery [4] é a linguagem de consulta padrão que está sendo elaborada pelo W3C e utiliza o mesmo modelo de dados apresentado na seção anterior. Trata-se de uma linguagem funcional na qual consultas são representadas como expressões. Numa consulta *XQuery* expressões têm a forma de variáveis, expressões de caminhos, chamada a funções, expressões condicionais, construtores de elementos, expressões FLWR(For-Let-Where-Return) etc., e podem ser aninhadas e combinadas usando operadores lógicos e aritméticos.

Para navegar através de caminhos na estrutura de um documento XML, *XQuery* usa expressões de "caminho" cuja sintaxe é uma abreviação da sintaxe *XPath*. O resultado do cálculo de uma expressão de caminho sobre um documento XML retorna uma floresta ordenada consistindo em nós que satisfazem a expressão e seus descendentes [9]. A ordem do resultado é ditada pela ordem dos elementos dentro do documento XML consultado.

Uma característica poderosa de *XQuery* é a presença de expressões FLWR. Uma expressão FLWR é usada sempre que é necessário interagir sobre elementos de uma coleção. As cláusulas *for-let* fazem variáveis interagirem sobre o resultado de uma expressão ou atribui o valor de expressões arbitrárias a variáveis. A cláusula *where* permite especificar restrições sobre essas variáveis. A cláusula *return* gera a saída da expressão FLWR, que pode ser um nodo, uma floresta ordenada de nodos, ou um valor primitivo. A cláusula *return* contém uma expressão geralmente composta por construtores de elementos, variáveis e sub-expressões aninhadas.

A seguir, apresentamos uma extensão da linguagem de consulta XQuery que permite atualizar dados XML.

3.3.3 Operadores para atualizar dados XML

Neste trabalho, utilizamos a extensão da linguagem de consulta XQuery proposta em [16]. Esta extensão corresponde à implementação dos operadores que são necessários para efetuarmos operações de atualização em documentos XML.

São definidos os seguintes operadores insert, delete e replace:

INSERT (conteúdo): Insere um conteúdo (o qual pode ser PCDATA, elemento, atributo ou referência) em um objeto selecionado na expressão de caminho.

DELETE ($filho): Se $filho é membro do objeto selecionado na expressão de caminho, ele será removido. Tipos válidos para para $filho incluem PCDATA, atributo, IDREF dentro de uma lista de IDREFS e elemento.

REPLACE ($filho, conteúdo): Operação atômica de substituição, equivalente a execução de um INSERT (conteúdo) seguido da execução de um DELETE ($filho).

O conjunto de operações apresentadas expressa logicamente atualizações XML. Em [16], essas operações são mapeadas para a sintaxe da linguagem XQuery. XQuery foi estendida com a seguinte estrutura para atualizações : FOR...LET... WHERE...UPDATE. Sendo especificada uma seqüência de suboperações na cláusula UPDATE:

```
FOR $L1  IN  ExpressãodeCaminho(Xpath)..,
LET $L := ExpressãodeCaminho(Xpath) ..,
WHERE predicado1,…..
UpdateOp
```

Figura 3.7: Extensão XQuery

Onde **UpdateOp** é definido da seguinte forma:

```
UPDATE $L {subOp {,subOp}*}
E subOp é:
  DELETE $filho |
  INSERT conteúdo (PCDATA | elemento| atributo) |
  REPLACE $filho WITH $conteúdo |
  FOR $L' IN SubExpressãodeCaminho(Xpath) ..,
        WHERE predicado2...,
        UpdateOp
```

Figura 3.8: Operadores da extensão XQuery

Considere o documento Bib.xml da Figura 3.1. Agora apresentamos um exemplo XQuery para cada uma das operações básicas:

- Inserção - Apresentamos o operador de inserção no exemplo da Figura 3.9, o qual insere um elemento "livro". O elemento "livro" contém os seguintes elementos filhos: "ano", "título", "isbn" e "editora", bem como, possui o atributo: "autoresref", o qual é uma referência para o elemento "autor" do tipo Tautor.

```
LET $a := document("bib.xml")/Bib
UPDATE $a {
INSERT
<livro autoresref="A1">
  <ano> 1999 </ano>
  <isbn> 3826561422 </isbn>
  <título> Transaction Management in Multidatabase Systems </título>
  <editora> Shaker-Verlag </editora>
 </livro> }
```

Figura 3.9: Inserção XQuery

- Remoção - Apresentamos o operador de remoção no exemplo da Figura 3.10, o qual apaga um elemento "livro" que possui o título "Transaction Management in Multidatabase Systems".

```
FOR $a IN document("bib.xml")/bib/livro
WHERE $a/titulo = "Transaction Management in
Multidatabase Systems"
UPDATE $a {
DELETE $a }
```

Figura 3.10: Remoção XQuery

- Modificação – Modificar o valor de um item tem o mesmo efeito que uma operação de remoção seguida de uma inserção, entretanto é útil possibilitar que uma atualização aconteça através de uma simples operação atômica. Assim, na Figura 3.8, apresentamos o operador de modificação **replace**, o qual modifica o valor do elemento "titulo" para "Transaction Management", onde o isbn do elemento "livro" é "3826561422".

```
FOR $a in document(“bib.xml”)/bib/livro[isbn=”3826561422”]
UPDATE $a {
REPLACE $a/titulo/text() WITH “Transaction Management” }
```

Figura 3.11: Modificação Xquery

Capítulo 4 - Atualização de Visões XML no XML Publisher

Neste Capítulo, apresentamos o processo de tradução de atualizações de visões XML no XML *Publisher*, um *framework* para publicação (consulta e atualização) de dados objeto-relacionais através de visões XML proposto em [24]. No XML *Publisher*, para cada visão XML publicada existe uma visão de objetos associada, chamada visão de objetos *default* (VOD), e consultas e atualizações sobre as visões XML são especificadas em *XQuery*. A tradução de consultas e atualizações *XQuery* no *framework* é facilitada; uma vez que a visão XML e sua VOD têm a mesma estrutura.

Este capítulo é organizado da seguinte forma. A Seção 4.1 contém uma visão geral do *framework,* com ênfase na tradução de atualizações; na Seção 4.2, descrevemos os passos para publicar visões XML no XML *Publisher*; na Seção 4.3, mostramos como atualizações *XQuery* são traduzidas no *framework*; na Seção 4.4, apresentamos como acessar o XML *Publisher* através de requisições http; na Seção 4.5, descrevemos o algoritmo para gerar o tipo da VOD a partir do esquema da visão XML.

4.1 Introdução

Para publicar uma visão XML no XML *Publisher*, o projetista deve definir uma visão de objetos denominada Visão de Objeto *Default* (VOD), cujos objetos têm a mesma estrutura dos elementos da visão XML, e os *"instead of triggers"* da VOD. No *framework* proposto, atualizações sobre a visão XML são traduzidas em atualizações sobre o esquema da visão de objeto. Essa tradução é simplificada, dado que os elementos da visão XML têm estrutura isomórfica à estrutura dos objetos da VOD. As visões de objeto são definidas sobre o esquema do banco de dados, ficando assim a cargo destas resolver o problema do mapeamento de tuplas de tabelas (relacionais ou objeto relacionais) em objetos de tipo complexo cuja estrutura é compatível com a estrutura do tipo dos elementos da visão XML.

Como foi discutido no Capítulo 2, através do mecanismo de visões de objeto conseguimos facilmente definir visões lógicas, ricamente estruturadas, sobre um banco de

dados já existente. No Oracle 9i, uma visão de objeto é definida através de uma consulta SQL3 que especifica como os objetos da visão são sintetizados a partir dos dados do banco de dados, e um *"instead of trigger"* é definido através de um bloco de comandos PL/SQL3 que especifica como atualizações definidas sobre a visão de objeto são traduzidas em atualizações especificadas sobre o banco de dados.

Nesta dissertação, apresentamos o processo de geração dos tradutores (*instead of triggers*) de atualizações das visões de objeto *default* e o Módulo de Tradução de de Atualização (MTA). O processo de geração de visões de objeto *default* e o Módulo de Processamento de Consultas (MPC) são discutidos em [35].

4.2 Ambiente para Publicação de Visões XML no XML Publisher

O XML *Publisher* oferece um ambiente para auxiliar o projetista no processo de publicação de visões XML. O processo de publicação compreende os seguintes passos: (i) especificação do esquema da visão XML; (ii) geração da respectiva VOD e *"instead of triggers"* da VOD; e (iii) configuração da visão XML no XML *Publisher*. Esses passos serão explicados nas seções a seguir.

4.2.1. Especificação do Esquema da visão XML

A tarefa de publicação no XML *Publisher* começa com o projetista definindo o esquema da visão XML. Consultas e atualizações *XQuery* sobre a visão XML serão definidas de acordo com esse esquema.

O esquema de cada visão XML deve ser especificado usando a linguagem XML *Schema*. Assim como uma visão de objetos contém um conjunto de objetos do mesmo tipo, uma visão XML no XML *Publisher* contém um conjunto de elementos do mesmo tipo. Sendo assim, o tipo do elemento raiz de cada visão XML definida no XML *Publisher* deve conter uma ou mais declarações de elementos multiocorrência ou monocorrência, como apresentado na figura 4.1:

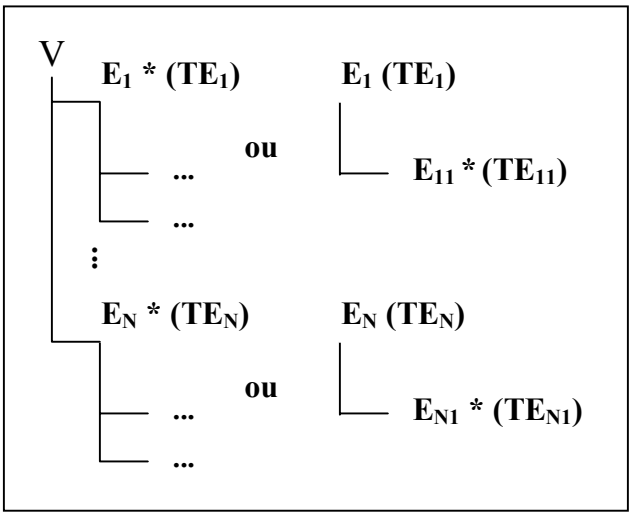


Figura 4.1: Tipo do elemento raiz.

Observe que o elemento raiz **V** é composto por elementos E_i do tipo T_{Ei}, $1 \leq i \leq n$. Como veremos a seguir, escolhe-se um elemento E_i que determina o nome da VOD. Caso o elemento E_i seja multiocorrência, a estrutura de T_{Ei} determina a estrutura da VOD. Caso contrario, T_{Ei} é um **tipo coleção**[2] e a estrutura da VOD será obtida a partir da estrutura do elemento filho de E_i.

No Ambiente de publicação do XML *Publisher*, o esquema da visão XML pode ser especificado com o auxílio de uma ferramenta gráfica. Inicialmente, a ferramenta solicita o nome do elemento raiz, o caminho para o arquivo que irá armazenar o esquema XML (Figura 4.2 (a)). Em seguida, o projetista deve especificar o tipo para cada elemento filho da raiz e seus elementos (Figura 4.2 (b)). Para cada elemento são pedidos seu tipo e sua cardinalidade. No caso de elementos de tipo complexo, então um novo tipo deve ser criado (nome e elementos).

[2] Um tipo coleção é um tipo complexo do esquema XML composto por uma única definição de elemento multiocorrência

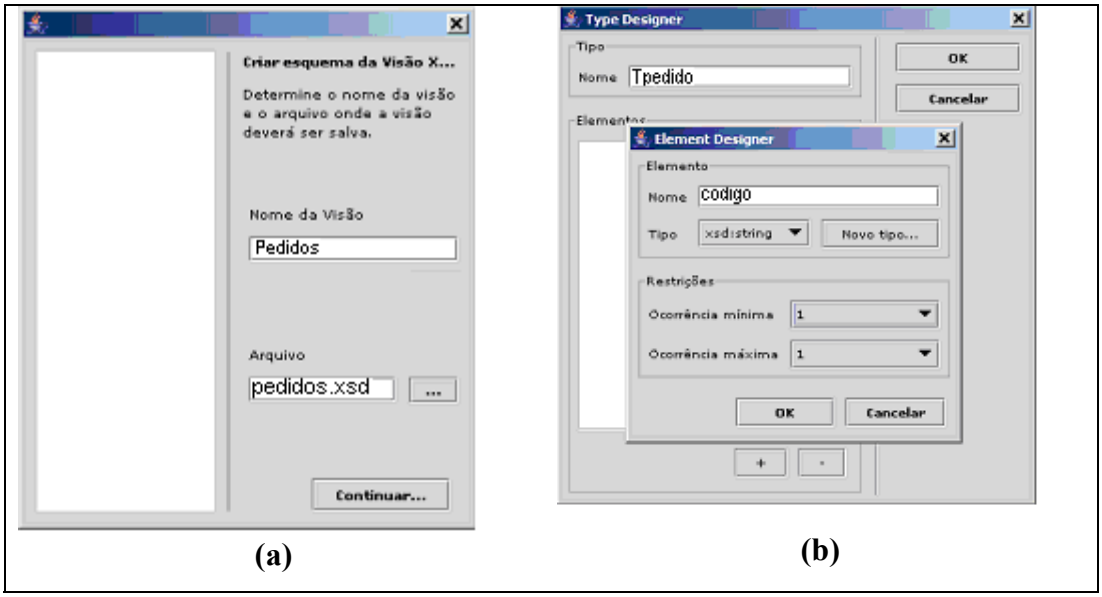


Figura 4.2: Especificando o esquema da visão XML.

4.2.2. Geração da VOD e *"instead of triggers"* da VOD

Após especificar o esquema da visão XML, o projetista segue definindo a visão de objetos *default* e os *"instead of triggers"* para a VOD. Como foi visto no Capítulo 2, para definir uma visão de objetos em um banco de dados objeto-relacional devemos criar os tipos da visão (esquema da visão) e uma consulta SQL3 que especifica como construir os objetos da visão a partir dos objetos do banco de dados. Então, o processo de geração da VOD e dos *"instead of triggers"* para a VOD consiste de três passos, como discutido a seguir.

***Passo 1*:** Geração do Esquema da VOD

Os tipos da VOD têm a mesma estrutura dos tipos complexos da visão XML. Considere, por exemplo, o esquema do banco de dados $\mathbf{B}_1$ apresentado na Figura 4.3. Suponha que o projetista deseja publicar a visão XML Pedidos definida sobre $\mathbf{B}_1$, cuja estrutura está definida na Figura 4.4 (a). O esquema da VOD de Pedidos é apresentado na Figura 4.4 (b). Observe que elementos monocorrência e multiocorrência da visão XML são diretamente mapeados em atributos monovalorados e multivalorados, respectivamente, da VOD. Por exemplo, o elemento multiocorrência listaItens é mapeado no atributo multivalorado listaItens, cujo tipo é composto por objetos do tipo $\mathbf{T}_{itens}$. Os elementos de listaItens são mapeados em atributos de $\mathbf{T}_{itens}$.

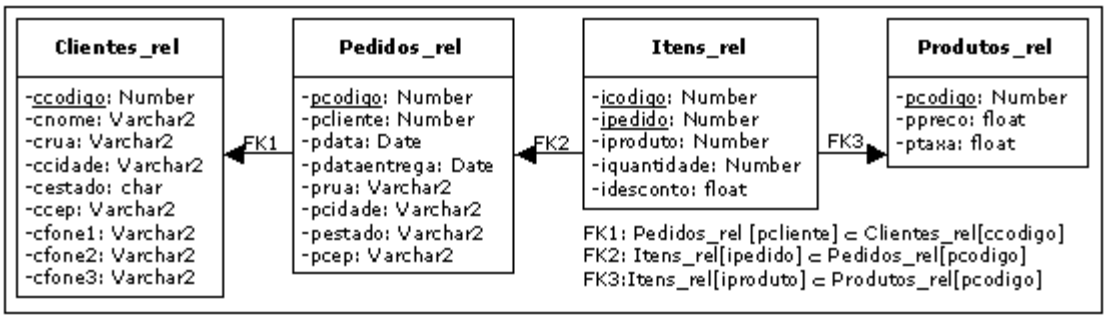


Figura. 4.3: Esquema do Bano de Dados Pedidos

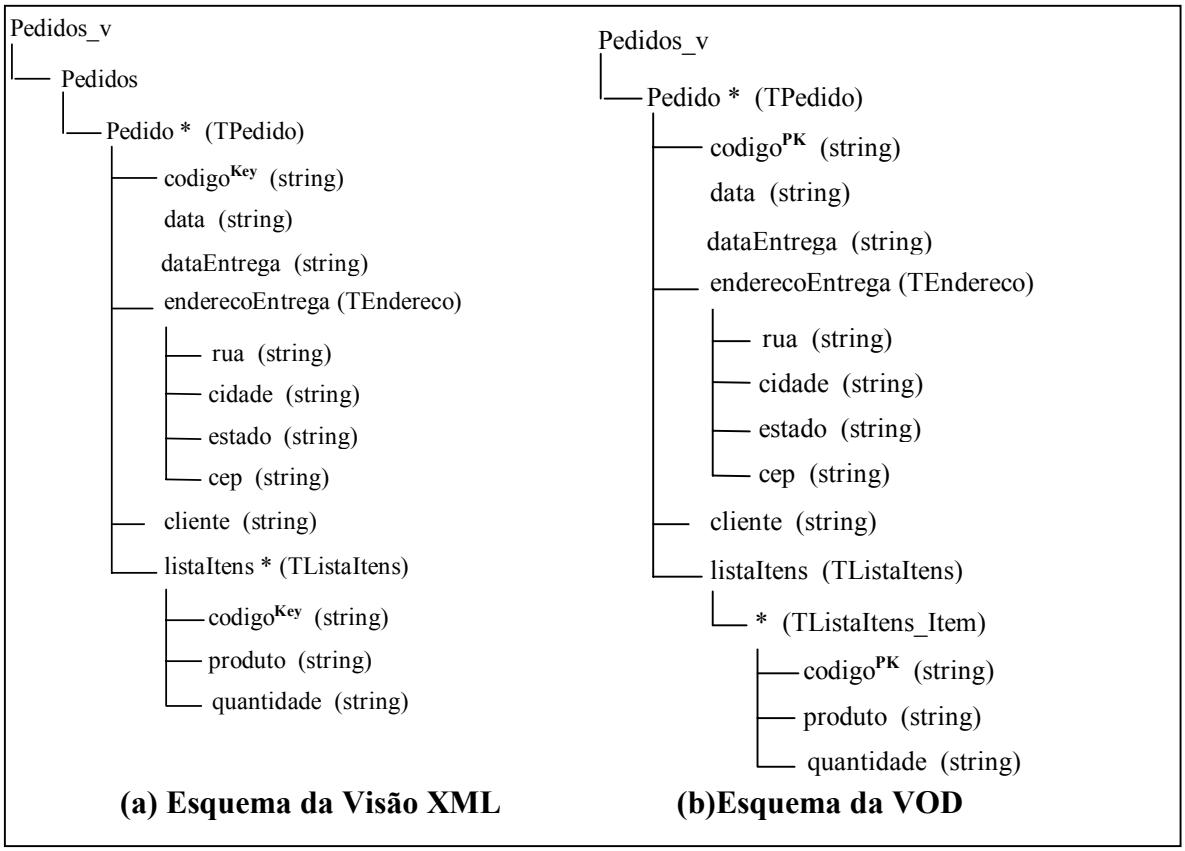


Figura 4.4: Esquemas da Visão Pedidos

No ambiente de publicação do XML *Publisher*, a ferramenta DSG (***D****efault Object View* ***S****chema* ***G****enerator*) gera automaticamente o esquema da VOD, com base no esquema XML da visão XML. O algoritmo implementado por essa ferramenta será apresentado na Seção 4.5.

Passo 2: Geração da consulta SQL3 da VOD

```
CREATE VIEW Pedidos_v OF Tpedido
WITH OBJECT IDENTIFIER (codigo) AS
   SELECT
Tpedido(tpedidos_rel•pcodigo,
        tpedidos_rel•pdata,
        tpedidos_rel•pdataEntrega,
        Tendereco ( tpedidos_rel•prua, tpedidos_rel•pcidade, tpedidos_rel•pestado, tpedidos_rel•pcep ),
        (SELECT tclientes_rel•cnome
         FROM Clientes_rel tclientes_rel
         WHERE tpedidos_rel•pcliente = tclientes_rel•ccodigo),
        CAST(MULTISET( SELECT
                          Titem(  titens_rel•icodigo,
                                  (SELECT tprodutos_rel•pnome
                                   FROM Produtos_rel tprodutos_rel
                                   WHERE titens_rel•iproduto= tprodutos_rel•pcodigo),
                                  titens_rel•iquantidade)
                       FROM Itens_rel  titens_rel ) AS Tlista_item )
FROM Pedidos_rel  tpedidos_rel
```

Figura 4.5: Definição da VOD Pedidos_v.

A Figura 4.5 apresenta a definição da VOD de Pedidos_v aplicada ao banco de dados B_1. A definição da VOD pode ser gerada de forma automática no ambiente de publicação do XML *Publisher*, através da ferramenta VBA (*View-By-Assertion*), que foi definida em [35].

Passo 3: Geração dos *"instead of triggers"* da VOD

Após especificar a definição da VOD, o projetista segue definindo os *"instead of triggers"* da VOD. A Figura 4.6 apresenta a definição do *"instead of trigger"* AdiçãoEmListaItens. Este trigger traduz inserções na coleção aninhada (NESTED TABLE) listaItens da visão Pedidos_v em inserções na tabelas Itens_rel. Observe que os valores dos atributos da tabela Itens_rel são obtidos a partir do objeto (:new) inserido na coleção aninhada listaItens e da visão Pedidos_v (:parent). Os *"instead of triggers"* podem ser gerados de forma semi-automática no ambiente de publicação do XML Publisher, através da ferramenta TAV (Tradutores de Atualização de Visão), a qual é apresentada no Capítulo 5. Com TAV, o projetista inicialmente carrega o esquema da VOD e o esquema do banco de dados e, através de uma GUI, define as assertivas de correspondência entre esses esquemas. Com base no conjunto de assertivas de correspondência da visão, TAV gera

automaticamente os *"instead of triggers"* da VOD de acordo com o mapeamento definido pelas assertivas.

```
CREATE TRIGGER AdicaoEmListaItens
INSTEAD OF INSERT ON NESTED TABLE listaItens OF Pedidos_v
BEGIN
INSERT INTO Itens_rel(icodigo, iquantidade, iproduto, ipedido)
       VALUES (:new.codigo,
               :new.quantidade,
               :new.produto,
               :parent.codigo);
END;
```

Figura. 4.6: Trigger de inserção na NT listaItens da VOD Pedidos_v

4.2.3. Configurações do XML *Publisher*

Após definir o esquema da visão XML, criar a consulta SQL3 da respectiva VOD e definir *"instead of triggers"* da VOD, devemos atualizar o arquivo de configuração do XML *Publisher* para efetivar a publicação. As configurações do XML *Publisher* são armazenadas em um documento XML de nome WXSConfig.xml. Esse documento contém informações que serão utilizadas pelo XML *Publisher* para o processamento de consultas e atualizações XQuery. Em [35], detalhes da configuração do XML Publisher são apresentados.

4.3 Disponibilizando o XML Publisher na Web

O *XML Publisher* é disponibilizado como uma aplicação web, de forma que seus serviços podem ser acessados por um cliente através de requisições *HTTP GET*. Essas requisições têm o seguinte formato: http://host/path?REQUEST=operação&param=value, onde host denota o endereço da aplicação *web* que hospeda o *XML Publisher*, path é o caminho da aplicação *XML Publisher* na aplicação *web* e REQUEST é o parâmetro da requisição que indica a operação submetida ao *XML Publisher*. O nome (param) e o valor (value) para o segundo parâmetro da requisição, variam de acordo com a operação. As operações permitidas no *XML Publisher* são:

1. ***GetCapabilities***: requisita os nomes das visões XML publicadas e ações suportadas (consultas e/ou atualizações) por estas.
2. ***GetSchema***: requisita o *XML Schema* de uma visão XML publicada.

3. ***ExecuteQuery***: requisita consultas XQuery definidas sobre o esquema da visão XML publicada.
4. ***ExecuteUpdate***: requisita atualizações XQuery definidas sobre o esquema da visão XML publicada.

Exemplo 4.1:

Seja $\mathbf{A}_1$ uma atualização XQuery definida para a visão XML **Pedidos**. Para executar essa atualização no XML *Publisher*, a aplicação *web* deve submeter a seguinte requisição ao XML *Publisher*:

http://www.lia.ufc.br/bd/xmlPublisher?REQUEST=ExecuteUpdate&QUERY= $\mathbf{A}_1$

Note que a atualização $\mathbf{A}_1$ foi passada como o valor do parâmetro QUERY. Observe que não foi preciso passar o nome da visão a ser atualizada, pois o nome da visão já está definido na própria atualização $\mathbf{A}_1$, como veremos no Capítulo 6.

4.3 Processamento de Atualizações no XML Publisher

Nesta seção, apresentamos uma visão geral sobre o processamento de atualizações *XQuery* no XML *Publisher*. Para realizar esse processamento, desenvolvemos um *software* chamado *XUpdateTranslator*, o qual será apresentado com mais detalhes no Capítulo 6.

Os passos para o processamento de atualizações XQuery no XML *Publisher* são apresentados na Figura 4.7. Seja **A** uma atualização XQuery aplicada a uma visão XML $\mathbf{X}_1$. Para processar **A** o XML *Publisher* procede da seguinte forma:

i. O *XUpdateTranslator* traduz **A** em uma atualização SQL3 **S** aplicada à respectiva VOD. Dado que a estrutura dos objetos da VOD tem a mesma estrutura dos elementos da visão XML, a tradução é facilitada e pode ser feita com base apenas no esquema da visão XML. Em seguida, **S** é submetido ao SGBD.
ii. A atualização **S** gerada é traduzida, pelos tradutores de atualizações (*instead of triggers*), em atualizações nas tabelas do banco de dados. No Capítulo 5, discutimos os algoritmos que geram os tradutores (*instead of triggers*) de atualização da visão de objetos.

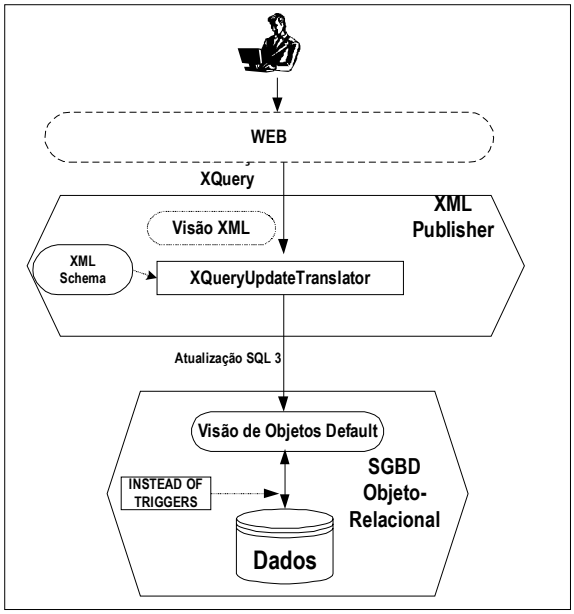


Figura 4.7: Processamento de atualizações no XML *Publisher*

Exemplo 4.2:

A seguir apresentamos um exemplo de processamento de uma atualização na visão Pedidos_v usando XML Publisher. Considere a atualização XQuery definida sobre a visão XML **Pedidos_v** (Figura 4.8). A atualização XQuery requisita a inserção do elemento `item` como filho dos nós $v2 satisfazendo as seguintes condições: $v2 ∈ $v1/listaItens onde $v1 ∈ view("Pedidos_v")/Pedidos/Pedido e $v1/cliente = "Wamberg" .

```
for $v1 in view("Pedidos_v")/Pedidos/Pedido,
 $v2 in $v1/listaItens
where $v1/cliente = "Wamberg"
update $v2
insert
  <item>
    <codigo>10</codigo>
    <produto>Impressora</produto>
    <quantidade>100</quantidade>
  </item>
```

Figura 4.8: Atualização XQuery A_1.

No *XUpdateTranslator* essa atualização XQuery é traduzida em uma seqüência de atualizações definidas sobre a VOD Pedidos_v mostrada na Figura 4.9. Dado que os tipos dos objetos da VOD Pedidos_v têm a mesma estrutura do tipo da visão XML, essa tradução é facilitada e pode ser realizada com base apenas no XML esquema da visão Pedidos_v.

As atualizações definidas sobre a VOD Pedidos_v são traduzidas pelo *"instead of trigger"* da Figura 4.6 em atualizações sobre o esquema do banco, sendo então processadas pelo SGBD.

```
DECLARE
COD_PED NUMBER;
CURSOR TabObjPai IS
        SELECT REF(v1)
        FROM Pedidos_v v1
       WHERE v1.cliente = 'Wamberg'
BEGIN
OPEN TabObjPai;
LOOP

  FETCH TabObjPai INTO COD_PER;
  EXIT WHEN PED_CUR%NOTFOUND;
  INSERT INTO  TABLE(  SELECT v1.listaItens
                       FROM Pedidos_v v1
                       WHERE REF(v1)=
COD_PED)
   VALUES(Titem('10','Impressora','100'))

END_LOOP;

END.
```

Figura 4.9: Atualização SQL A_2.

4.5 Algoritmo para Gerar o Esquema da VOD

Uma visão de objetos num banco de dados objeto-relacional é criada a partir do esquema da visão (tipos da visão) e sua definição, que é uma consulta sobre o Banco de Dados Objeto Relacional (BDOR) a qual especifica como os objetos da visão são gerados a partir do banco de dados. Neste trabalho discutiremos apenas como gerar o esquema da VOD. O problema de gerar a definição de visões de objetos é tratado em [35].

A Figura 4.10 apresenta o algoritmo **GeraEsquemaOR**, o qual gera o esquema de uma VOD a partir do esquema XML da visão XML. Inicialmente o algoritmo chama o procedimento **NomeiaTiposAnonimos** para transformar as definições de tipo anônimo do esquema XML em definições de tipos complexos nomeadas, a partir da declaração do elemento primário. Isso garante que todo tipo complexo do esquema XML seja mapeado em um tipo do esquema da VOD.

```
GeraEsquemaOR (S1: esquema da visão XML)
Início
φ = ∅ ;
NomeiaTiposAnonimos (S1)
SubstituiElementoMultiocorrencia ( S1 )
Para toda definição de tipo complexo T em S1, diferente do tipo do elemento raiz, faça:
      Caso 1: T é do TipoColeção
         Seja Ta o tipo do elemento A e n sua ocorrência máxima
         Seja P o elemento pai de A
         Se P é um elemento do tipo do elemento raiz/* Cria uma Nestead Table T de referências a Ta
                  φ = φ + Create Type T as Table of Ref Ta;
         Senão
                  Se Ta é uma referencia para um tipo /* Cria uma Nestead Table T de referências a Ta
                           φ = φ + Create Type T as Table of Ref Ta;
                  Senão
                           Se Ta possui um elemento E cujo tipo é um TipoColeção
                                    /* Cria uma Nestead Table T de referências a Ta
                                    φ = φ + Create Type T as Table of Ref Ta;
                           Senão
                                    Se  n é UNBOUNDED (Caso 1.2)/* Cria uma Nestead Table T de Ta
                                             φ = φ + Create Type T as Table of Ta;
                                    Senão /* Cria um VARRAY T de Ta com cardinalidade n
                                             φ = φ + Create Type T as VARRAY(n) of Ta;
                                    Fim-se  ;
                           Fim-se  ;
                  Fim-se;
         Fim-se;
      Caso 2: T não é do tipo coleção  /* Cria um tipo Ty com o mesmo nome de T
         φ = φ + "Create Type Ty as Object(
         Para cada elemento/atributo Ei, 1 ≤ i ≤ n, em T faça:
                  Seja Ai a declaração de atributo do esquema default de objetos associada ao elemento Ei
                  de T.
                  Seja Ti o tipo do elemento Ei e n sua ocorrência máxima.
                  Se Ti é um tipo referência (Caso 2.1)
                           Se n =1
                                    φ = φ + Ai REF Ti,
                           Senão ( n >1)
                                    Seja TC o TipoColeção que possui o elemento multiocorrência do tipo Ti
                                    φ = φ + Ai TC,
                           Fim_se;
                  Senão (Caso 2.2.)
                           φ = φ + Ai Ti,
                   Fim_Se;
         Fim_para;
         φ = φ + );
      Fim_se;
Fim_para;
Retorne(φ);}
/* Fim do Algoritmo GeraEsquemaOR */
```

Figura 4.10: Algoritmo **GeraEsquemaOR**.

O algoritmo segue com uma chamada ao procedimento SubstituiElementoMultiocorrencia (Figura 4.11) . Esse procedimento transforma as definições de

elementos multiocorrência do esquema da visão XML em definições de tipo coleção, quando necessário. Um tipo coleção é um tipo complexo do XML *Schema* composto por uma única definição de elemento multiocorrência. Essa transformação é necessária, pois um XML *Schema* só pode ser diretamente mapeado em um esquema objeto-relacional se todos os seus elementos multiocorrência são definidos como tipos coleção. Essa propriedade garante que os elementos multiocorrência definidos em um XML *Schema* sejam facilmente mapeados em *Nested Tables* ou *Varrays*, como indicado no Caso 1 do algoritmo GeraEsquemaOR.

Em seguida, o algoritmo GeraEsquemaOR transforma cada tipo complexo do esquema da visão XML em um tipo de objeto (tipo abstrato de dado - TAD) no esquema da VOD. Elementos monocorrência são transformados em atributos dos tipos da VOD. Elementos multiocorrência são descartados, uma vez que objetos numa *Nested Table* ou *Varray* não são nomeados.

```
1.  SubstituiElementoMultiocorrencia (S2 : esquema da visão XML)
2.  Inicio
3.  Para cada tipo complexo T do esquema S2 que não é um TipoColeção faça
4.
5.      Para cada elemento multiocorrência E de T faça
6.          Seja TE o tipo do elemento E
7.
8.          Se TE não é um tipo referência então
9.              Criar um tipo complexo TE_item, composto pelos elementos de TE;
10.             Em TE, substituir o seu conteúdo por um elemento multiocorrência
11.             E_item do tipo TE_item;
12.
13.             Em T, a cardinalidade do elemento E passa a ser monocorrência.
14.         Fim_se;
15.     Fim_para;
    Fim_para;
    Fim.
```

Figura 4.11: Procedimento **SubstituiElementoMultiocorrencia**.

Note que o esquema da visão XML é modificado apenas em caráter temporário, ou seja, as modificações servem apenas para gerar o esquema da VOD. Atualizações XQuery submetidas ao XML *Publisher* devem ser definidas sobre o XML *Schema* inicial. A seguir, apresentamos um exemplo para ilustrar o funcionamento desse algoritmo.

Exemplo 4.3:

Considere o esquema da visão XML Pedidos apresentado na Figura 4.4 (a). Para gerar o esquema da VOD de Pedidos, o algoritmo **GeraEsquemaOR** inicialmente chama o procedimento **NomeaTiposAnonimos**, mas como todos os tipos complexos a partir da declaração do elemento raiz do esquema da visão XML são possuem um nome, a chamada a esse procedimento não altera o esquema. Em seguida, chama o procedimento **SubstituiElementoMultiocorrencia**. Esse procedimento modifica a definição do tipo $\mathbf{T}_{pedido}$, pois o elemento **listaitens** desse tipo é multiocorrência e $\mathbf{T}_{pedido}$ não é de tipo coleção.

O procedimento **SubstituiElementoMultiocorrencia** cria um tipo complexo $\mathbf{T}_{listaitens\_item}$ composto pelos elementos do tipo complexo $\mathbf{T}_{listaitens}$; o tipo complexo $\mathbf{T}_{listaitens}$ será composto unicamente pelo elemento multiocorrência **listaitens_item** do tipo $\mathbf{T}_{listaitens\_item}$; a cardinalidade do elemento **listaitens** do tipo complexo $\mathbf{T}_{pedido}$ passa a ser monocorrência. Essas alterações no XML *Schema* da visão XML $\mathbf{X}_1$ são apresentadas na Figura 4.12.

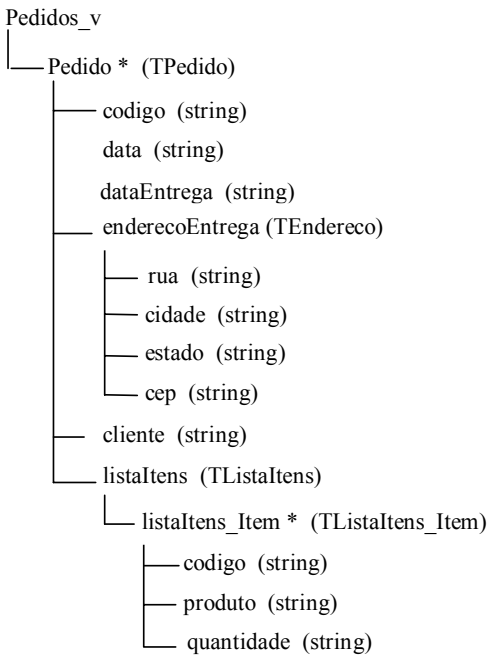


Figura 4.12: Esquema intermediário da visão XML **Pedidos_v**.

O próximo passo agora é gerar os tipos do esquema da VOD. Os quatro tipos do esquema da Figura 4.12 geram os quatro tipos de objeto no esquema da Figura 4.13. Os tipos $\mathbf{T}_{endereço}$, $\mathbf{T}_{Pedido}$ e $\mathbf{T}_{ListaItens\_item}$ são tratados pelo caso 2 do algoritmo, pois não são do tipo

coleção. Note que os elementos que compõem cada um desses tipos são mapeados em atributos no respectivo tipo de objeto no esquema da VOD. O tipo $T_{ListaItens\_item}$ é tratado pelo caso 1, pois é do tipo coleção. Este tipo é mapeado em um tipo Nested Table da VOD. Observe que o elemento desse tipo não é mapeado para o esquema da VOD.

```
create type Tendereco as object(
rua varchar(30),
cidade varchar(30),
estado varchar(30),
cep varchar(30)
);
/
create type TListaItens_Item as object(
codigo varchar(30),
produto varchar(30),
quantidade varchar(30)
);
/
create type TListaItens as table of
TListaItens_Item;
/
create type Tpedido as object(
codigo varchar(30),
data varchar(30),
dataEntrega varchar(30),
enderecoEntrega Tendereco,
cliente varchar(30),
listaItens TListaItens
);
```

Figura 4.13: Esquema da VOD Pedidos_v.

Capítulo 5 - Geração de Tradutores para Atualização de Bancos de Dados Relacionais através de Visões de Objeto

Neste capítulo, descrevemos os algoritmos que geram tradutores para as operações de atualização de visões de objetos em bancos de dados relacionais. Os algoritmos recebem como entrada: o esquema da visão, o esquema do banco de dados e as assertivas de correspondências da visão; e gera tradutores para as operações básicas de atualização de bancos de dados relacionais. Neste trabalho usamos assertivas de correspondências para especificar formalmente a correspondência entre o esquema da visão de objetos e o esquema relacional do banco de dados. Consideramos apenas as visões que preservam objetos. Uma visão preserva os objetos, quando cada objeto da visão é semanticamente equivalente a um objeto de uma tabela base. Os objetos o_1 e o_2 são semanticamente equivalentes ($o_1 \equiv o_2$) se e somente se representam uma mesma entidade do mundo real.

Este capítulo está organizado com se segue. Na Seção 5.1 apresentamos o processo de geração das assertivas de correspondências de uma visão de objetos. Na Seção 5.2, discutimos TAV, uma ferramenta gráfica para geração semi-automática dos tradutores de atualização de visões de objetos em bancos de dados relacionais. E, nas demais seções apresentamos os algoritmos que geram os tradutores para operações básicas de atualização de visões de objetos em bancos de dados relacionais.

5.1 Assertivas de Correspondências no Modelo Objeto-Relacional

Para gerar a definição dos tradutores de atualização de uma visão de objetos é necessário estabelecer as correspondências entre o esquema da visão e o esquema do banco de dados, ou seja, como as informações disponibilizadas em um dos esquemas estão relacionadas com o outro esquema. Nesta seção, apresentamos a definição formal dos vários tipos de assertiva de correspondência (AC) utilizada para especificar formalmente o relacionamento entre elementos de esquemas objeto-relacional. O formalismo proposto permite especificar várias formas de correspondência, inclusive casos onde os esquemas

possuem estruturas diferentes [54]. As AC's da visão de objetos são classificadas em três tipos: (i) AC de Extensão (ACE); (ii) AC de Caminho (ACC) e (iii) AC de Objeto(ACO). A seguir, apresentamos algumas definições necessárias para a definição formal dos tipos de assertiva citados acima.

5.1.1 Terminologias

Definição 5.1: Usamos o conceito de **ligação** para representar os relacionamentos entre tipos. Considere $\mathbf{T}_1$ e $\mathbf{T}_2$ tipos de um esquema. Existe uma ligação de $\mathbf{T}_1$ para $\mathbf{T}_2$ nas situações descritas a seguir:

i. (**Ligação de atributo de valor**) $\mathbf{T}_1$ contém um atributo cujo tipo é $\mathbf{T}_2$ ou coleção de objetos do tipo $\mathbf{T}_2$. Assim, $\mathbf{a}$: $\mathbf{T}_1 \rightarrow \mathbf{T}_2$ é uma ligação de $\mathbf{T}_1$ para $\mathbf{T}_2$. Dada uma instância $\mathbf{t}_1$ de $\mathbf{T}_1$, a expressão $\mathbf{t}_1 \bullet \mathbf{a}$ retorna o valor do atributo $\mathbf{a}$ (uma instância ou coleção de instâncias de $\mathbf{T}_2$).

ii. (**Ligação de atributo de referência**) $\mathbf{T}_1$ contém um atributo cujo tipo é uma referência ou uma coleção de referências para objetos do tipo $\mathbf{T}_2$. Assim, $\mathbf{a}$: $\mathbf{T}_1 \rightarrow \mathbf{T}_2$ é uma ligação de $\mathbf{T}_1$ para $\mathbf{T}_2$; Dada uma instância $\mathbf{t}_1$ de $\mathbf{T}_1$, a expressão $\mathbf{t}_1 \bullet \mathbf{a}$ retorna as instâncias de $\mathbf{T}_2$ referenciadas por $\mathbf{t}_1$ através do atributo $\mathbf{a}$.

iii. (**Ligação de chave estrangeira**) Existe uma chave estrangeira $\mathbf{fk}=\mathbf{R}_1[\mathbf{a}_1,..., \mathbf{a}_n] \subseteq \mathbf{R}_2[\mathbf{b}_1, ..., \mathbf{b}_n]$, onde $\mathbf{T}_1$ e $\mathbf{T}_2$ são os tipos[3] das tuplas de $\mathbf{R}_1$ e $\mathbf{R}_2$, respectivamente. Assim, $\mathbf{fk}$: $\mathbf{T}_1 \rightarrow \mathbf{T}_2$ é uma ligação de $\mathbf{T}_1$ para $\mathbf{T}_2$. Dada uma tupla $\mathbf{t}_1$ de $\mathbf{R}_1$, a expressão $\mathbf{t}_1 \bullet \mathbf{fk}$ retorna a tupla $\mathbf{t}_2$ de $\mathbf{R}_2$ tal que $\mathbf{t}_1 \bullet \mathbf{a}_i = \mathbf{t}_2 \bullet \mathbf{b}_i$, para $1 \leq i \leq n$.

iv. (**Inversa de ligação**) $\mathbf{T}_2$ tem uma ligação ℓ: $\mathbf{T}_2 \rightarrow \mathbf{T}_1$ tal que ℓ é uma ligação de atributo de referência ou de chave estrangeira. Então, a inversa da ligação ℓ, dada por ℓ^{-1}: $\mathbf{T}_1 \rightarrow \mathbf{T}_2$ é uma ligação de $\mathbf{T}_1$ para $\mathbf{T}_2$. Este tipo de ligação é chamado de **ligação virtual**. Os demais tipos de ligações são chamados de **ligação direta**.

[3] Para tratarmos de forma uniforme tabelas de objetos e tabelas de tuplas, assumimos que dada uma tabela de tuplas $\mathbf{R}$, $\mathbf{T}_R$ é o tipo das tuplas de $\mathbf{R}$. Este tipo é na verdade, definido implicitamente na própria criação da tabela $\mathbf{R}$.

Definição 5.2: Um objeto pode estar relacionado com outro objeto através da composição de duas ou mais ligações. Considere as ligações, ℓ_1: $\mathbf{T_1}\rightarrow\mathbf{T_2}$, ℓ_2: $\mathbf{T_2}\rightarrow\mathbf{T_3}$,..., ℓ_{n-1}: $\mathbf{T_{n-1}}\rightarrow\mathbf{T_n}$, onde $\mathbf{T_1}$, $\mathbf{T_2}$,..., $\mathbf{T_n}$ são tipos de um esquema objeto-relacional. Então, $\varphi = \ell_1\bullet\ell_2\bullet...\bullet\ell_{n-1}$ é um caminho de $\mathbf{T_1}$. Isso significa que as instâncias de $\mathbf{T_1}$ estão relacionadas com as instâncias de $\mathbf{T_n}$ através do caminho φ. Dada uma instância $\mathbf{t_1}$ de $\mathbf{T_1}$, a expressão $\mathbf{t_1}\bullet\varphi$ retorna objetos $\mathbf{t_n}$ de $\mathbf{T_n}$ tais que existem as instâncias $\mathbf{t_2}$, $\mathbf{t_3}$,..., $\mathbf{t_{n-1}}$ onde $\mathbf{t_i}$ está relacionada com $\mathbf{t_{i+1}}$ através da ligação ℓ_i, para $1\leq i< n$. Assim, o tipo do caminho φ é $\mathbf{T_n}$ ($\mathbf{T_\varphi}=\mathbf{T_n}$). Se as ligações ℓ_1, ℓ_2,..., ℓ_{n-1} são monovaloradas, então φ é um **caminho monovalorado**, caso contrário φ é um **caminho multivalorado**. Se ℓ_{n-1} é uma ligação de referência então φ é um **caminho de referência**, caso contrário φ é um **caminho de valor**.

5.1.2 Assertivas de Correspondência de Extensão

A extensão de uma tabela (ou visão) é o conjunto de objetos que são membros da tabela (ou visão) em um determinado instante. As Assertivas de Correspondência de Extensão (ACE) especificam a correspondência existente entre a extensão da visão com a extensão de sua tabela pivô [55]. No resto desta seção, sejam **V** uma visão, **S** um esquema OR e **R** uma tabela em **S**. Existem dois tipos de ACEs, definidas a seguir.

Definição 5.3: A ACE $[\mathbf{V}] \equiv [\mathbf{R}]$ especifica que, dados σ_s o estado de **S** e σ_v a extensão de **V** em σ_s, $\mathbf{t_v} \in \sigma_v$ sss $\exists \mathbf{t_R} \in \sigma_s(\mathbf{R})$ tal que $\mathbf{t_v}\equiv\mathbf{t_R}$.

Definição 5.4: A ACE $[\mathbf{V}] \equiv [\mathbf{R}[\mathbf{P}]]$, onde **P** é um predicado condicional, especifica que, dados σ_s o estado de **S** e σ_v a extensão de **V** em σ_s, $\mathbf{t_v} \in \sigma_v$ sss $\exists \mathbf{t_R} \in \sigma_s(\mathbf{R})$ tal que $\mathbf{t_R}$ satisfaz a condição **P** e $\mathbf{t_v}\equiv\mathbf{t_R}$.

5.1.3 Assertivas de Correspondência de Caminho

No restante desta seção, considere $\mathbf{T_v}$ e $\mathbf{T_b}$, onde $\mathbf{T_v}$ e $\mathbf{T_b}$ são semanticamente relacionados, no contexto de uma ACE ou ACC. As ACC's podem ser de três tipos como definidas a seguir:

Definição 5.5: Seja **c** um atributo de $\mathbf{T_v}$ e φ um caminho de $\mathbf{T_b}$, onde a cardinalidade de **c** é igual à cardinalidade de φ. A ACC $[\mathbf{T_v}\bullet\mathbf{c}] \equiv [\mathbf{T_b}\bullet\varphi]$ especifica que para quaisquer instâncias

t_v de T_v e t_b de T_b, se $t_v \equiv t_b$, então $t_v \bullet c \equiv t_b \bullet \varphi$. No caso em que **c** é monovalorado, dados $t'_v = t_v \bullet \varphi_v$ e $t'_b = t_b \bullet \varphi$, então $t'_v \equiv t'_b$. No caso em que **c** é multivalorado, temos que t'_v pertence a $t_v \bullet \varphi_v$ sss existe t'_b em $t_b \bullet \varphi$, tal que $t'_v \equiv t'_b$.

Definição 5.6: Suponha $b_1, b_2, ..., b_n$ atributos de valor atômico de T_b e **a** um atributo de T_v, cujo tipo T_a contém atributos atômicos $a_1, a_2, ..., a_n$. A ACC $[T_v \bullet a, \{a_1, a_2, ..., a_n\}] \equiv [T_b, \{b_1, b_2, ..., b_n\}]$ especifica que dada uma instância t_v de T_v e t_b de T_b, se $t_v \equiv t_b$, então $t_v \bullet a \bullet a_i = t_b \bullet b_i$, para $1 \leq i \leq n$.

Definição 5.7: Seja **a** um atributo multivalorado de valor atômico de T_v e φ um caminho monovalorado de T_b, de tipo T_φ. Sejam $c_1, c_2, ..., c_n$ atributos monovalorados de valores atômicos de T_φ. A ACC $[T_v \bullet a] \equiv [T_b \bullet \varphi, \{c_1, c_2, ..., c_n\}]$ especifica que para quaisquer instâncias t_v de T_v e t_b de T_b, se $t_v \equiv t_b$, então t'_v pertence à $t_v \bullet a$ sss existe um objeto t'_b em $\{t_b \bullet \varphi \bullet c_1, t_b \bullet \varphi \bullet c_2, ..., t_b \bullet \varphi \bullet c_n\}$, tal que $t'_v = t'_b$.

5.1.4 Assertivas de Correspondência de Objeto

As assertivas de correspondências de objeto (ACO) especificam sob que condições dois objetos, que são instâncias de tipos semanticamente relacionados, representam o mesmo objeto do mundo real, ou seja, são semanticamente equivalentes.

Definição 5.8: Considere T_v um tipo do esquema da visão e T_b um tipo de uma tabela base, os quais são semanticamente relacionados. Sejam $v_1, v_2, ..., v_n$ atributos de T_v e $b_1, b_2, ..., b_n$ atributos de T_b. A ACO $\psi:[T_v, \{v_1, v_2, ..., v_n\}] \equiv [T_b, \{b_1, b_2, ..., b_n\}]$ especifica que para qualquer instância t_v de T_v e t_b de T_b, se $t_v \bullet v_i = t_b \bullet b_i$, $1 \leq i \leq n$, então $t_v \equiv t_b$.

5.1.5 Uso de Assertivas de Correspondência para Especificar Visões de Objetos

Neste trabalho, uma visão de objeto **V** sobre o esquema **S** é definida por uma 4-tupla $\mathbf{V} = < T_v, \psi_v, \mathscr{C}_V, \mathscr{A}_V >$ onde T_v é o tipo dos objetos de **V**, ψ_v é uma ACE, $\mathscr{C}_V$ é um conjunto de ACC e $\mathscr{A}_V$ é um conjunto de ACO. Isso define um mapeamento funcional, denotado DEF_V, de estados σ_s de **S** em estados da visão, como definido a seguir.

Caso 1: Para ψ_v da forma [**V**] ≡ [**R**[**P**]], temos:

$$DEF_V(\sigma_s) = \{ \text{Construtor}\mathbf{T_v}\_\mathbf{T_R}(\mathbf{t}) \mid \mathbf{t} \in \sigma_s(\mathbf{R}) \text{ e } \mathbf{P}(\mathbf{t})=\text{true} \},$$

onde Construtor$\mathbf{T_v}_\mathbf{T_R}$ (**t**) cria uma instância **v** de $\mathbf{T_V}$ tal que **v≡t**. Note que se **v** ≡ **t** então **v** deve satisfazer a todas as ACC's de $\mathbf{T_v}$ & $\mathbf{T_b}$. Assim, os valores dos atributos do objeto **v** criado são definidos de acordo com as assertivas de correspondência de caminho de $\mathbf{T_v}$ & $\mathbf{T_R}$. Em [35] é apresentada uma ferramenta que, a partir das ACCs da visão, gera automaticamente o construtor de objetos da visão.

Caso 2: Para ψ_v da forma [**V**] ≡ [**R**], temos:

$$DEF_V(\sigma_s) = \{ \text{Construtor}\mathbf{T_v}\_\mathbf{T_R}(\mathbf{t}) \mid \mathbf{t} \in \sigma_s(\mathbf{R}) \}.$$

Exemplo 5.1:

No resto desta seção, considere a visão **Pedidos_v** cujos objetos são do tipo $\mathbf{T_{pedido}}$ (vide Figura 5.1) e o esquema do banco de dados **Pedidos_rel,** apresentado na Figura 5.2. Considere que a ACE de **Pedidos_v** é dada por ψ_1: [**Pedidos_v]** ≡ [**Pedidos_rel].** ψ_1 especifica que as extensões da visão **Pedidos_v** e da tabela **Pedidos_rel** denotam o mesmo conjunto de objetos do mundo real. As ACC's de **Pedidos_v** são mostradas na Figura 5.3

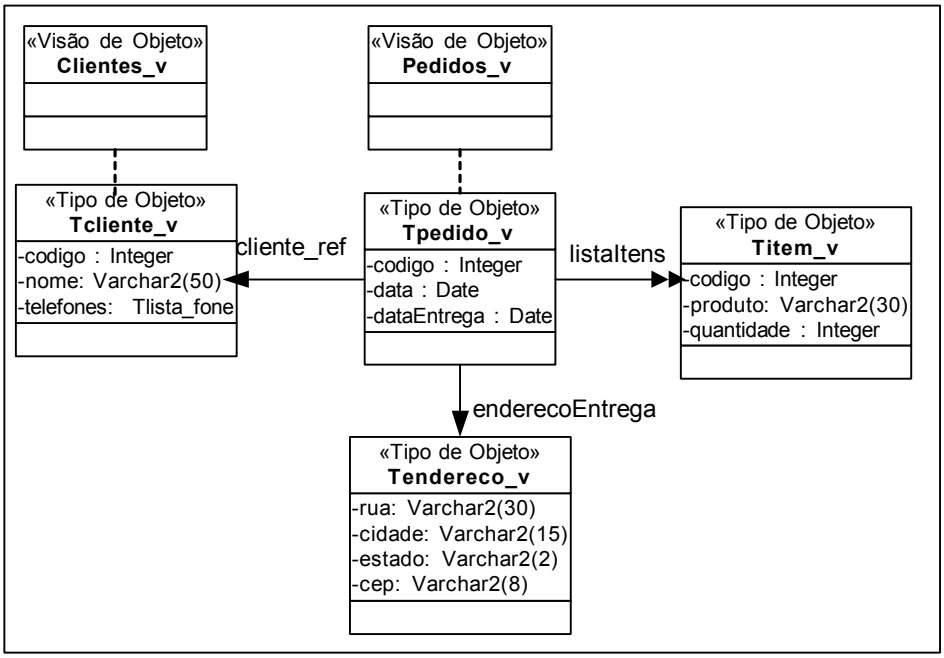


Figura 5.1: Esquema da Visão de Objetos **Pedidos_v**.

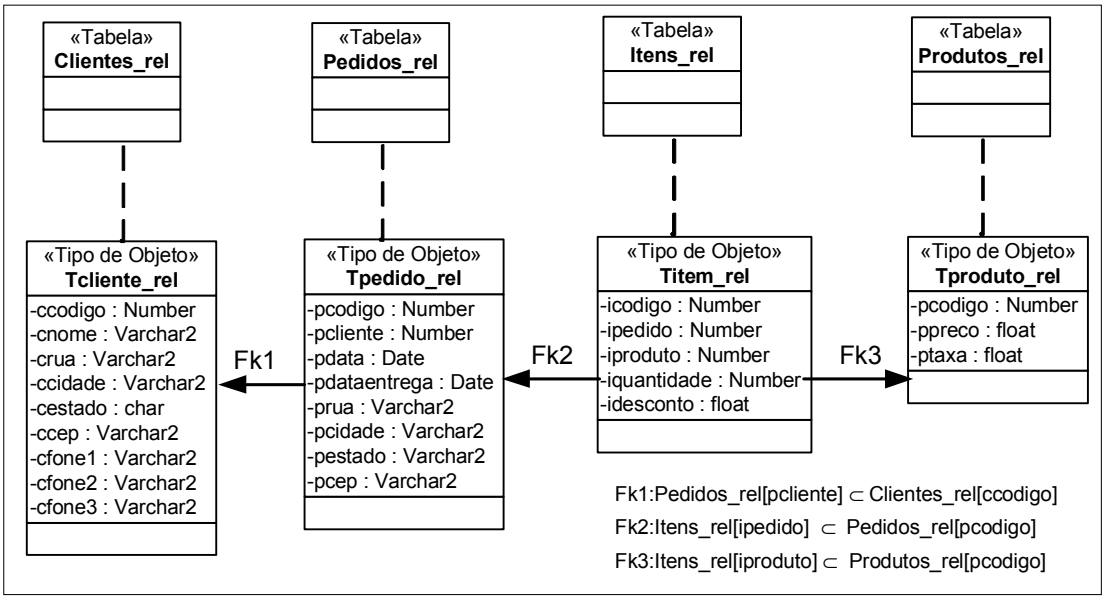


Figura 5.2: Esquema Relacional do Banco de Dados Pedidos_rel.

O processo de geração das assertivas é *top down* e recursivo. Primeiro definimos as ACC's dos atributos de $\mathbf{T}_{pedido\_v}$ com atributos/caminhos da tabela pivô **Pedidos_rel**. No caso de atributos de referência ou de tipos complexos deve-se então recursivamente definir as ACC's dos seus atributos com o seu tipo base.

Das ACC's de $\mathbf{T}_{pedido\_v}$&$\mathbf{T}_{pedidos\_rel}$ temos que dada uma instância $\mathbf{t}_v$ de $\mathbf{T}_{pedido\_v}$, e uma instância $\mathbf{t}_b$ de $\mathbf{T}_{pedidos\_rel}$ tal que $\mathbf{t}_v \equiv \mathbf{t}_b$ então:

(i) $\mathbf{t}_v \bullet codigo = \mathbf{t}_b \bullet pcodigo$ (de $\psi 3$)

(ii) $\mathbf{t}_v \bullet data = \mathbf{t}_b \bullet pdata$ (de $\psi 4$)

(iii) $\mathbf{t}_v \bullet dataEntrega = \mathbf{t}_b \bullet pdataEntrega$ (de $\psi 5$)

(iv) $\mathbf{t}_v \bullet enderecoEntrega = \mathbf{T}_{endereco\_v}(\mathbf{t}_b \bullet prua, \mathbf{t}_b \bullet pcidade, \mathbf{t}_b \bullet pestado, \mathbf{t}_b \bullet pcep)$ (de $\psi 6$)

(v) $\mathbf{t}_v \bullet cliente\_ref \equiv \mathbf{t}_b \bullet \mathbf{Fk}_1$ (de $\psi 7$). Como o tipo de cliente_ref é uma referência para $\mathbf{T}_{cliente\_v}$, que é um tipo estruturado, deve-se definir as ACC's de **T**cliente_v & **T**clientes_rel.

(vi) $\mathbf{t}' \in \mathbf{t}v \bullet listaItens$ sss existe $\mathbf{t} \in \mathbf{t}_b \bullet \mathbf{Fk}_2^{-1}$ e $\mathbf{t} \equiv \mathbf{t}'$ (de $\psi 8$). Como listaItens é uma coleção de objetos de tipo $\mathbf{T}_{item\_v}$, que é um tipo estruturado, deve-se definir as ACC's de **T**item_v & **T**itens_rel.

ACC's de T_{pedido} & $T_{pedidos\_rel}$

ψ_3: $[T_{pedido\_v} \bullet codigo] \equiv [T_{pedidos\_rel} \bullet pcodigo]$

ψ_4: $[T_{pedido\_v} \bullet data] \equiv [T_{pedidos\_rel} \bullet pdata]$

ψ_5: $[T_{pedido\_v} \bullet dataEntrega] \equiv [T_{pedidos\_rel} \bullet pdataEntrega]$

ψ_6: $[T_{pedido\_v} \bullet enderecoEntrega], \{rua, cidade, estado, cep\}] \equiv [T_{pedidos\_rel}, \{prua, pcidade], pestado, pcep\}]$

ψ_7: $[T_{pedido\_v} \bullet cliente\_ref] \equiv [T_{pedidos\_rel} \bullet fk_1]$

ψ_8: $[T_{pedido\_v} \bullet listaItens] \equiv [T_{pedidos\_rel} \bullet fk_2^{-1}]$

ACC's de T_{item_v} & T_{itens_rel}

ψ_{10}: $[T_{item\_v} \bullet codigo] \equiv T_{itens\_rel} \bullet icodigo$

ψ_{11}: $[T_{item\_v} \bullet produto] \equiv [T_{itens\_rel} \bullet iproduto]$

ψ_{12}: $[T_{item\_v} \bullet quantidade] \equiv [T_{itens\_rel} \bullet iquantidade]$

ACC's de $T_{cliente\_v}$ & $T_{clientes\_rel}$

ψ_{14}: $[T_{item\_v} \bullet codigo] \equiv T_{itens\_rel} \bullet ccodigo$

ψ_{15}: $[T_{item\_v} \bullet nome] \equiv [T_{itens\_rel} \bullet cnome]$

ψ_{16}: $[T_{item\_v} \bullet telefones] \equiv [T_{itens\_rel}, \{cfone1, cfone2, cfone3\}]$

Figura 5.3: ACC's de **Pedidos_v**

Das ACC's de T_{item_v} & T_{itens_rel} temos que dada uma instância t_v de T_{item_v}, e uma instância t_b de T_{itens_rel} tal que $t_v \equiv t_b$ então:

(i) $t_v \bullet codigo = t_b \bullet icodigo$ (de ψ_{10})

(ii) $t_v \bullet produto = t_b \bullet iproduto$ (de ψ_{11})

(iii) $t_v \bullet quantidade = t_b \bullet iquantidade$ (de ψ_{12})

Das ACC's de $T_{cliente\_v}$ & $T_{clientes\_rel}$ temos que dada uma instância t_v de $T_{cliente\_v}$, e uma instância t_b de $T_{clientes\_rel}$ tal que $t_v \equiv t_b$ então:

(i) $t_v \bullet codigo = t_b \bullet ccodigo$ (de ψ_{14})

(ii) $t_v \bullet nome = t_b \bullet cnome$ (de ψ_{15})

(iii) $t_v \bullet telefones = T_{lista\_fone}(t_b \bullet cfone1, t_b \bullet cfone2, t_b \bullet cfone2)$ (de ψ_{16})

As assertivas de correspondência de objeto de **Pedidos_v** são mostradas na Figura 5.4. As ACO's são inferidas a partir das ACC's e das chaves primárias das tabelas do banco de dados. Por exemplo, da ACC ψ_3 e da chave primária da tabela **Pedidos_rel**, inferimos a ACO ψ_2: $[T_{pedido\_v}, \{codigo\}] \equiv [T_{pedido\_rel}, \{pcodigo\}]$ que especifica que para quaisquer instâncias t_v de T_{pedido_v} e t_b de T_{pedido_rel}, se $t_v \bullet codigo = t_b \bullet pcodigo$, então $t_v \equiv t_b$.

Como definido em [35], a definição SQL3 de uma visão de objetos pode ser gerada automaticamente a partir das ACs da visão. A Figura 5.5 mostra, a título de ilustração, a definição SQL3 da Visão de Objetos **Pedidos_v** que realiza o mapeamento definido pelas Assertivas. Como mostraremos na Seção 5.3, os tradutores de atualização (*instead of triggers*) de **Pedidos_v** podem ser gerados automaticamente a partir das ACs da visão.

ACO de T_{pedido_v} & $T_{pedidos\_rel}$
ψ_2: [T_{pedido_v}, {codigo}] ≡ [$T_{pedidos\_rel}$, {pcodigo}]

ACO de T_{Item_v} & T_{itens_rel}
ψ_9 [T_{item_v},{codigo}] ≡ [T_{itens_rel},{icodigo}]

ACO de $T_{cliente\_v}$ & $T_{clientes\_rel}$
ψ_{13}:[$T_{cliente\_v}$, {codigo}] ≡ [$T_{clientes\_rel}$, {ccodigo}]

Figura 5.4: ACO's de **Pedidos_v**.

```
CREATE VIEW Pedidos_v OF T_pedido
WITH OBJECT IDENTIFIER (codigo) AS
   SELECT
1. T_pedido(                                                    Construtor_T_pedido-T_pedidos_rel
2.        t_pedidos_rel•pcodigo,                                                    (de ψ3 )
3.        t_pedidos_rel•pdata,                                                      (de ψ4 )
4.        t_pedidos_rel•pdataEntrega,                                               (de ψ5 )
5.        T_endereco ( t_pedidos_rel•prua, t_pedidos_rel•pcidade, t_pedidos_rel•pestado, t_pedidos_rel•pcep ),  (de ψ6 )
6.        (SELECT REF (t_cliente)                                                   (de ψ7 )
          FROM Clientes_v t_cliente, Clientes_rel t_clientes_rel
          WHERE t_pedidos_rel•pcliente = t_clientes_rel•ccodigo
                t_clientes_rel•ccodigo = t_cliente•codigo),                         (da ACO ψ9)
7.        CAST(MULTISET( SELECT                                                     (de ψ8 )
                                                        Construtor_T_item-T_itens_rel
                  T_item(
8.                      t_itens_rel•icodigo,                                      (de ψ10)
9.                      (SELECT t_produtos_rel•pcodigo                            (de ψ11)
                         FROM Produtos_rel t_produtos_rel
                         WHERE t_itens_rel•iproduto= t_produtos_rel•pcodigo),
10.                     t_itens_rel•iquantidade)                                  (de ψ12)
                FROM Itens_rel t_itens_rel
11.       ) AS T_lista_item )

FROM Pedidos_rel t_pedidos_rel
```

Figura 5.5: Definição da visão de objetos **Pedidos_v**

5.2 Geração de Tradutores de Atualização com TAV

Nesta seção, apresentamos TAV (*Tradutor de Atualização de Visão*), um protótipo de uma ferramenta semi-automáica para gerar tradutores de atualização (instead of triggers) para visões de objetos no Oracle 9i [15]. Como estudo de caso, considere o esquema do banco de dados **Pedidos_rel** apresentado na Figura 5.2 e a visão de objeto **Pedidos_v** (Figura 5.1), cujos objetos são do tipo $\mathbf{T}_{\text{pedido}}$.

A seção começa com o usuário definindo o nome da visão e o nome do seu tipo. Em seguida, o usuário deve criar os atributos do tipo da visão (Figura 5.6 (a)). Para isso, usa a tela de criação de atributo (Figura 5.6 (b)). Além do nome, são pedidos o tipo do atributo, sua cardinalidade e se o mesmo é de valor ou de referência. No caso de atributos de tipo estruturado, então um novo tipo deve ser criado (nome e atributos). No caso de atributos de referência, deve-se definir seu escopo (visão de objeto referenciada). Se a visão ainda não existe, deverá ser criada. Quando utilizamos TAV na geração dos tradutores de atualização (instead of triggers) de visões de objeto *default* para o XML *Publisher*, a definição dos atributos do tipo da visão é feita automaticamente pela ferramenta GeraEsquemaVOD, como foi mostrado no Capítulo 4.

Após definir os atributos do tipo da visão, o usuário deve definir as assertivas de correspondência da visão. Primeiro, o usuário deve definir a ACE. Para isso, o usuário seleciona, de uma lista de tabelas e visões do banco de dados, a tabela base ou visão base, e, se necessário, define a condição de seleção. Para a visão **Pedidos_v** a tabela **Pedidos_rel** é selecionada e a ACE ψ_1:[**Pedidos_v**] ≡ [**Pedidos_Rel**] é gerada.

Em seguida, o usuário deve realizar o *matching* [56, 57, 58] do tipo da visão com o tipo base. Para isso, TAV exibe uma tela contendo, em formato de árvore de diretório, a estrutura do tipo da visão e a estrutura do tipo base (Figura 5.6 (c)). Observe que para o tipo base, além dos atributos, são mostradas as demais ligações do tipo (ligações de chave estrangeira e inversas), de modo que o usuário possa definir caminhos que naveguem por essas ligações. Para definir a ACC de um atributo da visão, o usuário relaciona graficamente o atributo da visão com um atributo ou caminho do tipo da tabela base. A seguir mostramos como gerar as ACC's para alguns dos atributos do tipo $\mathbf{T}_{\text{pedido}}$:

– Para definir a ACC do atributo codigo, o usuário seleciona codigo no esquema da visão e pcodigo no esquema do banco de dados. Ao clicar no botão “Salvar”, TAV mostra a ACC gerada (ψ_3: [T_{pedido}•codigo] ≡ [$T_{pedidos\_rel}$•pcodigo]);

– Para definir a ACC do atributo listaItens, o usuário seleciona listaItens no esquema da visão e fk_2^{-1} no esquema do banco de dados. Observe que para o atributo listaItens o “*” denota que o atributo é multivalorado e T_{item}, especificado entre parênteses, é o tipo dos objetos em listaItens. Como T_{item} é um tipo estruturado, o usuário deve então definir recursivamente as correspondências para os atributos de T_{item}.

– Para definir a ACC do atributo enderecoEntrega, o qual não tem correspondência direta com nenhum atributo do tipo base, o usuário seleciona o atributo enderecoEntrega e a tabela base **Pedidos_rel**, e, então, salva. Em seguida, o usuário define a correspondência para cada atributo de $T_{endereco}$ com os atributos de $T_{pedidos\_rel}$. A ACC gerada é ψ_6:[T_{pedido}•enderecoEntrega,{rua, cidade, estado, cep}] ≡ [$T_{pedidos\_rel}$, {prua, pcidade, pestado, pcep}].

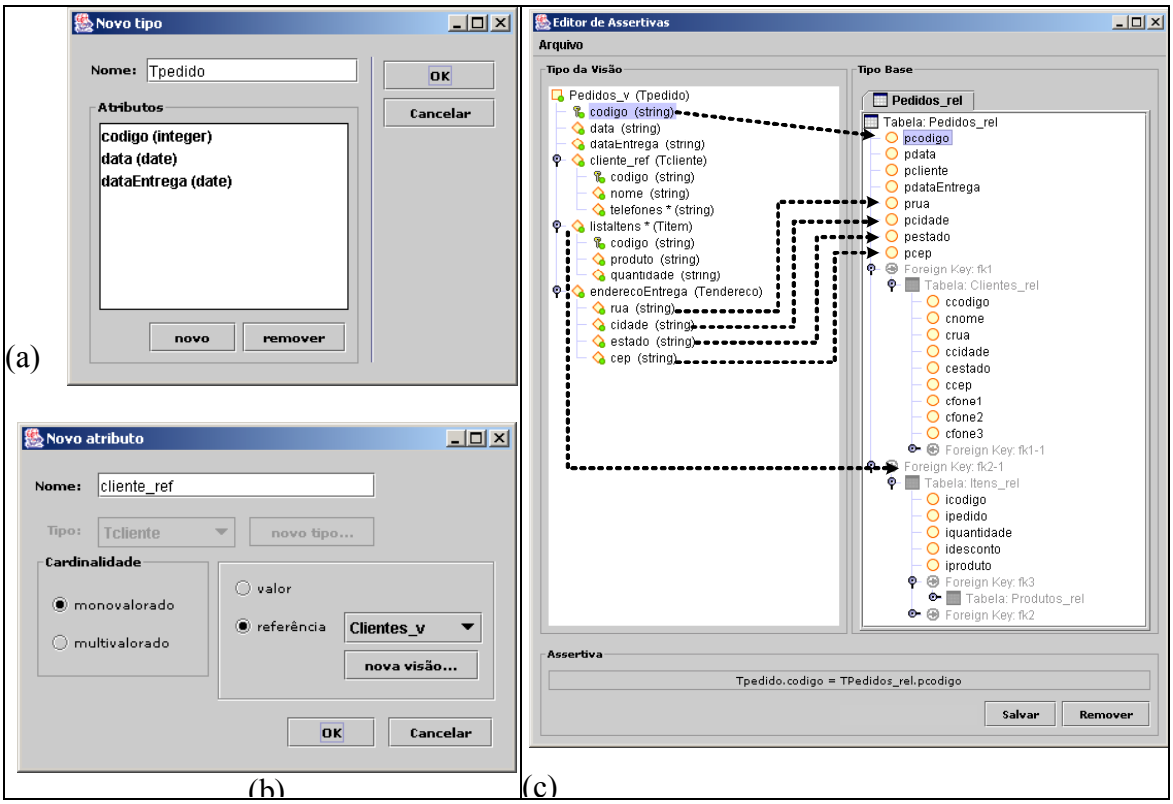


Figura 5.6: Telas do TAV: (a) Editor de Tipo; (b) Editor de Atributo; (c) Editor de Assertivas do TAV.

As ACO's de **Pedidos_v** são inferidas a partir das ACC's e das chaves das tabelas. Por exemplo, ψ_2 é inferida de ψ_3 e do fato de que pcodigo é a chave primária da tabela **Pedidos_rel**. Caso um atributo da chave primária não tenha um correspondente na visão, TAV resolve o problema, adicionando o atributo à ACC correspondente, e, por fim, à ACO.

A geração automática da definição SQL3 da visão de objetos é discutida em [35]. A Figura 5.5 apresenta, a título de ilustração, a definição da visão de objetos **Pedidos_v**.

Após definir as assertivas da visão, os tradutores de atualização podem ser gerados automaticamente. A Figura 5.7 apresenta o exemplo de um tradutor de atualização para a operação de inserção na coleção aninhada **listaitens** da visão de objetos **Pedidos_v**.

```
CREATE OR REPLACE TRIGGER AdicãoEmListaItens
INSTEAD OF INSERT ON NESTED TABLE listaItens OF Pedidos_v
BEGIN
 Insert Into Itens_rel

 ( icodigo, iquantidade, iproduto, ipedido )
  values ( :new.codigo,                          (de ψ10)
           :new.quantidade,                      (de ψ12)
            :new.produto,                        (de ψ11)
             :parent.código                      (da ACO ψ9)
          );

END;
```

Figura 5.7: Tradutor AdicãoEmListaItens

Em [37], foram definidos os algoritmos que geram tradutores de atualização para bancos de dados objeto-relacionais através de visões de objetos em uma linguagem de alto nível, isto é, de forma indepedente de implementação. Também foram definidas as condições em que as traduções são possíveis, isto é, onde não existe ambigüidade no nível de dados.

Neste trabalho, utilizamos os resultados apresentados em [37] e geramos os tradutores de atualizacao (*instead of triggers*) para o Oracle 9i. Utilizamos o enfoque apresentado em [37] para definir quais os tipos de tradutores podem ser gerados.

Nas seções a seguir discutimos os algoritmos para as operações básicas de atualização na visão de objetos. Os algoritmos recebem como entrada: o esquema da visão, o esquema do banco de dados e as assertivas de correspondências da visão; e gera tradutores (*instead of triggers*) para as operações básicas de atualização de banco de dados relacional.

Neste trabalho, desenvolvemos os algoritmos que geram os tradutores de atualização para as seguintes operações básicas de atualização:

- Adição de um objeto em uma coleção aninhada de uma visão;
- Remoção de um objeto de uma coleção aninhada de uma visão;
- Modificação de atributos monovalorados de uma visão;
- Adição de um objeto em uma visão;
- Remoção de um objeto de uma visão.

5.3 Definição de Tradutores para Operações de Adição em Coleções Aninhadas

Nesta seção apresentamos o algoritmo para geração dos tradutores para a operação de adição em coleções aninhadas. No Oracle9i o comando INSERT pode ser usado para adicionar objetos em coleções aninhadas. Por exemplo, considere a visão **Pedidos_v** apresentada na Figura 5.5. Para adicionar um item na coleção aninhada **listaltens** do pedido da visão **Pedidos_v**, cujo código é 1, podemos usar a atualização mostrada na Figura 5.8. No Oracle 9i, **:new** refere-se ao objeto inserido na coleção aninhada, e **:parent** refere-se ao objeto "pai" do objeto inserido; que no exemplo é o pedido de código=1 da visão **Pedidos_v**. Note que a subconsulta "TABLE(SELECT v•**listaltens** FROM **Pedidos_v** v WHERE v•codigo =1)" retorna a coleção aninhada do objeto pai selecionado. A condição na cláusula WHERE deve garantir que um único objeto é selecionado.

```
INSERT INTO TABLE( SELECT v•listaltens FROM Pedidos_v v WHERE v•codigo =1)
(codigo,produto,quantidade)
 VALUES(1,2,10);
```

Figura 5.8: Exemplo de inserção na coleção aninhada **listaltens** da visão **Pedidos_v**

No resto desta seção, considere a visão **V** cujos objetos são do tipo T_v e **lista_**T_c um atributo multivalorado (coleção aninhada) de T_v, cujo valor é um conjunto de objetos do tipo T_c, como mostrado na Figura 5.9 (a). Suponha que: (i) a extensão da visão **V** é definida por uma ACE de equivalência $V \equiv R_b$ ou de subconjunto $V \subset R_b$, onde R_b é a tabela base cujos objetos são do tipo T_{Rb} e (ii) A ACC de **lista_**T_c é dada por: $[T_v \bullet \text{lista\_}T_c] \equiv [T_{Rb} \bullet \varphi]$, onde

$\varphi=\ell_1 \bullet \ell_2 \bullet . \bullet \ell_n$ é um caminho de $\mathbf{T}_{Rb}$ (Figura 5.9 (b)), com ligações ℓ_1: $T_{Rb} \rightarrow \mathbf{T}_{R\ell 1}$, l_i: $\mathbf{T}_{R\ell_{i-1}} \rightarrow \mathbf{T}_{R\ell_i}$, $2 \leq i \leq n$.

No caso em que esquema do banco de dados é relacional, no caminho $\varphi=\ell_1 \bullet \ell_2 \bullet \ldots \bullet \ell_n$, temos que ℓ_1, ..., e ℓ_n são ligações de chave estrangeira ou inversas de chave estrangeira. No resto desta seção, suponha que ℓ_1 é uma ligação de chave estrangeira dada por $\mathbf{R}_b [\mathbf{f}_1^{\ell 1}, ..., \mathbf{f}_{m_1}^{\ell 1}] \subset \mathbf{R}_{\ell 1} [\mathbf{g}_1^{\ell 1},.., \mathbf{g}_{m_1}^{\ell 1}]$ ou inversa da chave estrangeira dada por $\mathbf{R}_{\ell 1} [\mathbf{g}_1^{\ell 1}, ..., \mathbf{g}_{m_1}^{\ell 1}] \subset \mathbf{R}_b [\mathbf{f}_1^{\ell 1}, ..., \mathbf{f}_{m_1}^{\ell 1}]$; e ℓ_i , $2 \leq i \leq n$, é uma ligação de chave estrangeira dada por $\mathbf{R}_{\ell_{i-1}} [\mathbf{f}_1^{\ell_i}, ..., \mathbf{f}_{m_i}^{\ell_i}] \subset \mathbf{R}_{\ell_i} [\mathbf{g}_1^{\ell_i},.., \mathbf{g}_{m_i}^{\ell_i}]$ ou inversa da chave estrangeira dada por $\mathbf{R}_{\ell_i} [\mathbf{g}_1^{\ell_i}, ..., \mathbf{g}_{m_i}^{\ell_i}] \subset \mathbf{R}_{\ell_{i-1}} [\mathbf{f}_1^{\ell_i}, ..., \mathbf{f}_{m_i}^{\ell_i}]$.

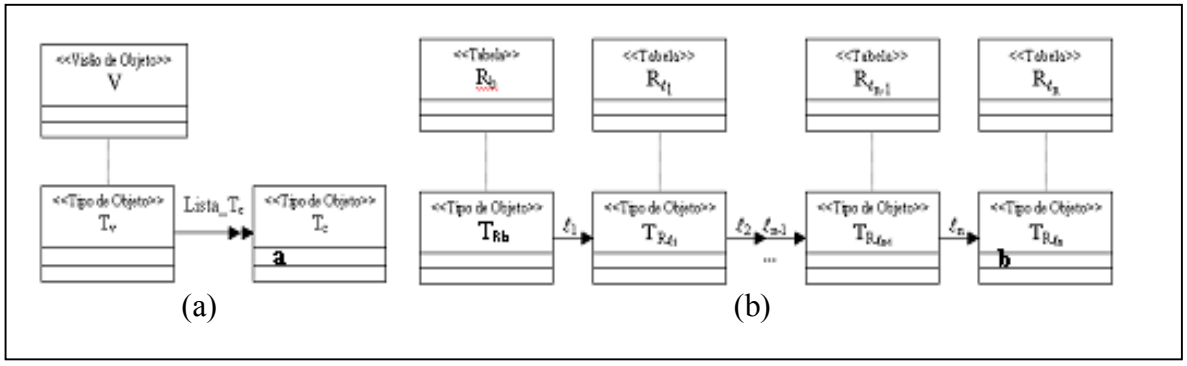


Figura 5.9: Coleção aninhada lista_T_c (a) e Caminho $\varphi = \ell_1 \bullet .. \bullet \ell_{n-1} \bullet \ell_n$ de $\mathbf{T}_{Rb}$ (tipo da tabela base $\mathbf{R}_b$) (b)

O algoritmo gera tradutores para os dois casos descritos a seguir. Em ambos os casos, só poderá existir uma única ligação multivalorada no caminho de derivação de lista_T_c (φ), uma vez que, como mostrado em [37], na existência de mais de uma ligação ocorrerá ambigüidade no nível de dados. O Caso 1 trata a situação em que a ligação multivalorada é a ultima ligação de φ, e ,o Caso 2, a situação em que a ligação multivalorada não é a ultima. Mostraremos que nestes casos não existe ambigüidade no nível de dados e a tradução pode ser gerada em tempo de projeto.

Caso 1: Ligação multivalorada é a última

Neste caso, como mostrado em [37], para garantirmos que não existe ambigüidade no nível de dados, as seguintes condições devem ser satisfeitas,:

1. As ligações $\ell_1,..,\ell_{n-1}$ e suas inversas são ligações monovaloradas e a ligação ℓ_n é multivalorada. Desta forma a ligação multivalorada é a última ligação do caminho e cada objeto de $\mathbf{R_b}$ está relacionado com um único objeto de $\mathbf{R}_{\ell_{n-1}}$ através do caminho $\ell_1 \bullet \ell_2 \bullet ... \bullet \ell_{n-1}$ e vice-versa.
2. As assertivas de correspondência de caminhos entre $\mathbf{T_c}$ e $\mathbf{T}_{R_{\ell_n}}$ podem ser de um dos tipos definidos abaixo:
   2.1. $[\mathbf{T_c \bullet a}] \equiv [\mathbf{T}_{R_{\ell_n}} \bullet \mathbf{b}]$, onde **a** é um atributo monovalorado de $\mathbf{T_c}$ e **b** é um atributo de valor atômico de $\mathbf{T}_{R_{\ell_n}}$;
   2.2. $[\mathbf{T_c \bullet a}, \{\mathbf{a_1}, \mathbf{a_2}, ..., \mathbf{a_n}\}] \equiv [\mathbf{T}_{R_{\ell_n}}, \{\mathbf{b_1}, \mathbf{b_2}, ..., \mathbf{b_n}\}]$, onde **a** um atributo de $\mathbf{T_c}$, cujo tipo $\mathbf{T_a}$ contém atributos atômicos $\mathbf{a_1}, \mathbf{a_2}, ..., \mathbf{a_n}$ e $\mathbf{b_1}, \mathbf{b_2}, ..., \mathbf{b_n}$ atributos de valor atômico de $\mathbf{T}_{R_{\ell_n}}$;
   2.3. $[\mathbf{T_c \bullet a}] \equiv [\mathbf{T}_{R_{\ell_n}}, \{\mathbf{b_1}, \mathbf{b_2}, ..., \mathbf{b_n}\}]$, onde **a** é um atributo multivalorado de valor atômico e $\mathbf{b_1}, \mathbf{b_2}, ..., \mathbf{b_n}$ atributos de valor atômico de $\mathbf{T}_{R_{\ell_n}}$;

   Para os casos 2.4 e 2.5 a seguir, considere $\varphi' = \ell'_1 \bullet \ell'_2 ... \bullet \ell'_p$ um caminho de $\mathbf{T}_{R_{\ell_n}}$, onde $\boldsymbol{\ell'_1}$ é uma chave estrangeira de R_{ℓ_n} dada por $R_{\ell_n}[\mathbf{f}_1^{\ell'1}, ..., \mathbf{f}_{m_1}^{\ell'1}] \subset R'_{\ell_1} [\mathbf{g}_1^{\ell'1}, .., \mathbf{g}_{m_1}^{\ell'1}]$, e ℓ'_i, é uma ligação de chave estrangeira dada por $\mathbf{R'}_{\ell'_{i-1}} [\mathbf{f}_1^{\ell'i}, ..., \mathbf{f}_{m_i}^{\ell'i}] \subset \mathbf{R'}_{\ell'_i} [\mathbf{g}_1^{\ell'i}, .., \mathbf{g}_{m_i}^{\ell'i}]$ ou inversa da chave estrangeira dada por $\mathbf{R'}_{\ell'_i} [\mathbf{g}_1^{\ell'i}, ..., \mathbf{g}_{m_i}^{\ell'i}] \subset \mathbf{R'}_{\ell'_{i-1}} [\mathbf{f}_1^{\ell'i}, ..., \mathbf{f}_{m_i}^{\ell'i}]$, $2 \leq i \leq p$, tal que: ℓ'_1 é uma ligação monovalorada direta cuja inversa pode ser multivalorada ou monovalorada. As ligações $\ell'_2 . \bullet .. \bullet \ell'_p$ e suas inversas são monovaloradas. Desta forma, temos que cada objeto de $\mathbf{R'}_{\ell'_1}$ está relacionado com um único objeto de $\mathbf{R'}_{\ell'_p}$ e vice-versa;
   2.4. $[\mathbf{T_c \bullet a}] \equiv [\mathbf{T}_{R_{\ell_n}} \bullet \varphi' \bullet \mathbf{k}]$, onde **a** é um atributo de valor atômico de $\mathbf{T_c}$ e **k** é um atributo e identificador de $\mathbf{R'}_{\ell_p}$. Esta condição garante que uma seleção em $\mathbf{R'}_{\ell_p}$, dado um valor de **k**, retorna uma única instância de $\mathbf{R'}_{\ell_p}$**;**

2.5. $[T_c \bullet \mathbf{a}] \equiv [T_{R_{\ell_n}} \bullet \varphi']$, onde **a** é um atributo de valor estruturado ou de referência do tipo T_a.

```
CREATE OR REPLACE TRIGGER AdicãoEm lista_Tc_Caso1
INSTEAD OF INSERT ON NESTED TABLE lista_Tc OF V
[DECLARE ]
BEGIN
[UTL_REF_SELECT OBJETC( )]
/* É uma função do Oracle utilizada para "desreferenciar" um objeto*/
INSERT INTO R_ℓn
( b1, ..., bn)
 values (Qb1, ..., Qbn);
```

Figura 5.10: Tradutor para Adição Em lista_T_c (Caso 1)

Figura 5.10 mostra o molde do tradutor "AdicãoEm lista_T_c" para este caso. De acordo com o tradutor, um pedido de adição de um objeto (:new) do tipo T_c na coleção lista_T_c do objeto pai (:parent) é traduzido em uma inserção da tupla $t_{novo} = (V_{b1},... V_{bn})$ na tabela R_{ℓ_n}. O valor do atributo b_i (V_{bi}), $1 \leq i \leq n$, de t_{novo} é definido pela subconsulta Q_{bi}, a qual computa o valor do atributo b_i a partir do objeto inserido (:new) e seu pai (:parent). Q_{bi} é definida com base nas ACs de T_c & $T_{R_{\ell_n}}$ e na ACC de **lista_T_c** (T_v.**lista_T_c** $\equiv T_b.\varphi$), de acordo com os casos definidos para o procedimento **GVA**$_1$ (**G**era **V**alor de **A**tributo) na Tabela 5.1 abaixo:

Caso 1: **b** é um atributo da chave estrangeira na ligação ℓ_n do caminho φ ($[T_v \bullet \text{lista\_}T_c] \equiv [T_{R_b} \bullet \ell_1 \bullet \ ... \ \bullet \ell_n]$)

Caso 1.1: φ possui somente uma ligação ℓ_1 ($\varphi = \ell_1$) inversa da chave estrangeira dada por
$R_{\ell_1}[g_1^{\ell_1}, ..., g_{m_1}^{\ell_1}] \subset R_b[f_1^{\ell_1}, ..., f_{m_1}^{\ell_1}]$ e $b = g_q^{\ell_1}$, $1 \leq q \leq m_1$, e
a ACO de T_{R_b} com T_v é dada por: $[T_{R_b},\{ f_1^{\ell_1}, ..., f_{m_1}^{\ell_1}\}] \equiv [T_v, \{d_1,..,d_{m_1}\}]$.

Q_b := :parent• d_q

Caso 1.2: φ possui mais de uma ligação (n>1) e $b = g_q^{\ell_n}$, $1 \leq q \leq m_n$, onde $f_q^{\ell_n}$ é parte da chave estrangeira
$R_{\ell_{n-1}}[f_1^{\ell_n}, ..., f_{m_n}^{\ell_n}] \subset R_{\ell_n}[g_1^{\ell_n},.., g_{m_n}^{\ell_n}]$ e a ACO de T_{R_b} com T_v é dada por: $[T_{R_b},\{ f_1^{\ell_1}, ..., f_{m_1}^{\ell_1}\}] \equiv [T_v, \{d_1,.., d_{m_1}\}]$.

Q_b:= **Select** $t_{R_{\ell_{n-1}}} \bullet f_q^{\ell_n}$
From R_{R_b} t_{R_b}, R_{ℓ_1} $t_{R_{\ell_1}}$,….., $R_{\ell_{n-1}}$ $t_{R_{\ell_{n-1}}}$
Where $t_{R_b} \bullet f_q^{\ell_1} = t_{R_{\ell_1}} \bullet g_q^{\ell_1}$, $1 \leq q \leq m_1$ **and** $t_{R_{\ell_{j-1}}} \bullet f_q^{\ell_j} = t_{R_{\ell_j}} \bullet g_q^{\ell_j}$, $1 \leq q \leq m_j$, $2 \leq j \leq n$
and $t_{R_b}.f_1^{\ell_1}$ =:parent•d_1 **and** ... **and** $t_{R_b} \bullet f_m^{\ell_1}$=:parent•$d_{m_1}$

Caso 2: **b** monovalorado de valor atômico e ψ:$[T_c \bullet a] \equiv [T_{R_{\ell_n}} \bullet b]$, onde **a** é um atributo monovalorado de T_c
Q_b:=:new•**a** ;

Caso 3: **b**=**d**i, **d**i é monovalorado de valor atômico e ψ: $[T_c \bullet a] \equiv [T_{R_{\ell_n}}, \{d_1,d_2,...,d_w\}]$, onde **a** é um atributo multivalorado de T_c do tipo varray
Q_b :=:new•**a[i]** ;

Caso 4: $\mathbf{b} = \mathbf{d}i$, $\mathbf{d}i$ é monovalorado de valor atômico e ψ:[$\mathbf{T_c}\bullet\mathbf{a}$,{$\mathbf{a}_1,\mathbf{a}_2,...,\mathbf{a}_w$}]≡[$\mathbf{T}_{R_{\ell_n}}$,{$\mathbf{d}_1,\mathbf{d}_2,...,\mathbf{d}_w$}]

$\mathbf{Q}_b$:=:new•**a**•$\mathbf{a}_i$;

Para os casos 5 e 6 considere $\varphi' = \ell'_1\bullet\ell'_2...\bullet\ell'_p$ um caminho de $\mathbf{T}_{R_{\ell_n}}$, onde $\boldsymbol{\ell}'_1$ é uma chave estrangeira de R_{ℓ_n} dada por $R_{\ell_n}[\mathbf{f}_1^{\ell'_1}, ..., \mathbf{f}_{m_1}^{\ell'_1}] \subset R'_{\ell_1}[\mathbf{g}_1^{\ell'_1},.., \mathbf{g}_{m_1}^{\ell'_1}]$, e ℓ'_i, é uma ligação de chave estrangeira dada por $\mathbf{R'}_{\ell'_{i-1}}[\mathbf{f}_1^{\ell'_i}, ..., \mathbf{f}_{m_i}^{\ell'_i}] \subset \mathbf{R'}_{\ell'_i}[\mathbf{g}_1^{\ell'_i},.., \mathbf{g}_{m_i}^{\ell'_i}]$ ou inversa da chave estrangeira dada por $\mathbf{R'}_{\ell'_i}[\mathbf{g}_1^{\ell'_i}, ..., \mathbf{g}_{m_i}^{\ell'_i}] \subset \mathbf{R'}_{\ell'_{i-1}}[\mathbf{f}_1^{\ell'_i}, ..., \mathbf{f}_{m_i}^{\ell'_i}]$, $2 \le i \le p$;.

Caso 5: **b** é um atributo da ligação $\boldsymbol{\ell}'_1$ e $[\mathbf{T_c}\bullet\mathbf{a}] \equiv [\mathbf{T}_{R_{\ell_n}}\bullet\varphi'\bullet\mathbf{k}]$ e $\mathbf{b} = \mathbf{f}_q^{\ell'_1}$, $1 \le q \le m_1$.

$\mathbf{Q}_b$:= **Select** $t_{R'\ell'_1}\bullet\mathbf{g}_q^{\ell'_1}$

From $\mathbf{R'}_{\ell'_1}\ t_{R'\ell'_1},....., \mathbf{R'}_{\ell'_p}\ t_{R'\ell'_p}$

Where $t_{R'\ell'_{j-1}}\bullet\mathbf{f}_k^{\ell'_j} = t_{R'\ell'_j}\bullet\mathbf{g}_k^{\ell'_j}$, $1 \le k \le m_j$, $2 \le j \le p$ **and** $t_{R'\ell'_p}\bullet\mathbf{k}$= :**new•a**

Caso 6: **b** é um atributo da ligação $\boldsymbol{\ell}'_1$ e $[\mathbf{T_c}\bullet\mathbf{a}] \equiv [\mathbf{T}_{R_{\ell n}}\bullet\varphi']$.

Caso 6.1: **a** é um atributo de valor estruturado do tipo $\mathbf{T_a}$

Caso 6.1.1: $\boldsymbol{\varphi}'$ possui somente uma ligação $\boldsymbol{\ell}'_1$ ($[\mathbf{T_c}\bullet\mathbf{a}] \equiv [\mathbf{T}_{R_{\ell_n}}\bullet\ell'_1]$), $\mathbf{b} = \mathbf{f}_q^{\ell'_1}$, $1 \le q \le m_1$, e ACO de $\mathbf{T}_{R'\ell'_1}$ com $\mathbf{T_a}$ é dada por: $[\mathbf{T_a},\{\mathbf{h}_1,..,\mathbf{h}_{m_1}\}] \equiv [\mathbf{T}_{R'\ell'_1}, \{\mathbf{g}_1^{\ell'_1},.., \mathbf{g}_{m_1}^{\ell'_1}\}]$

$\mathbf{Q}_b$:= :new•**a**•$\mathbf{h}_q$

Caso 6.1.2: $\boldsymbol{\varphi}'$ possui mais que uma ligação (p>1) e $\mathbf{b} = \mathbf{f}_q^{\ell'_1}$, $1 \le q \le m_1$ e ACO de $\mathbf{T}_{R'\ell'_p}$ com $\mathbf{T_a}$ é dada por: $[\mathbf{T_a},\{\mathbf{h}_1,..,\mathbf{h}_m\}] \equiv [\mathbf{T}_{R'\ell'_p}, \{\mathbf{e}_1,..,\mathbf{e}_m\}]$

$\mathbf{Q}_b$:= **Select** $t_{R'\ell'_1}\bullet\mathbf{g}_q^{\ell'_1}$

From $\mathbf{R'}_{\ell'_1}\ t_{R'\ell'_1},....., \mathbf{R'}_{\ell'_p}\ t_{R'\ell'_p}$

where $t_{R'\ell'_{j-1}}\bullet\mathbf{f}_k^{\ell'_j} = t_{R'\ell'_j}\bullet\mathbf{g}_k^{\ell'_j}$, $1 \le k \le m_j$, $2 \le j \le p$ **and** $t_{R'\ell'_p}\bullet\mathbf{e}_i$= :**new•a•**$\mathbf{h}_i$>, $1 \le i \le m$

Caso 6.2: **a** é um atributo de referência do tipo $\mathbf{T_a}$

/* Seja $**a** o objeto referenciado por :new.**a**. Nesse caso é realizada a chamada da função UTL_REF_SELECT OBJETC($**a**, :new.**a**) no tradutor gerado. UTL_REF_SELECT OBJETCT é uma função do Oracle utilizada para "desreferenciar" um objeto, então temos que a variável $**a** do tipo $\mathbf{T_a}$ recebe o objeto da referência "**:new.a**";*/

Caso 6.2.1: $\boldsymbol{\varphi}'$ possui somente uma ligação $\boldsymbol{\ell}'_1$ ($[\mathbf{T_c}\bullet\mathbf{a}] \equiv [\mathbf{T}_{R_{\ell_n}}\bullet\ell'_1]$), $\mathbf{b} = \mathbf{f}_q^{\ell'_1}$, $1 \le q \le m_1$, e ACO de $\mathbf{T}_{R'\ell'_1}$ com $\mathbf{T_a}$ é dada por: $[\mathbf{T_a},\{\mathbf{h}_1,..,\mathbf{h}_{m_1}\}] \equiv [\mathbf{T}_{R'\ell'_1}, \{\mathbf{g}_1^{\ell'_1},.., \mathbf{g}_{m_1}^{\ell'_1}\}]$

$\mathbf{Q}_b$:= :$**a**•$\mathbf{h}_q$

Caso 6.2.2: $\boldsymbol{\varphi}'$ possui mais que uma ligação (p>1) e $\mathbf{b} = \mathbf{f}_q^{\ell'_1}$, $1 \le q \le m_1$ e ACO de $\mathbf{T}_{R'\ell'_p}$ com $\mathbf{T_a}$ é dada por: $[\mathbf{T_a},\{\mathbf{h}_1,..,\mathbf{h}_m\}] \equiv [\mathbf{T}_{R'\ell'_p}, \{\mathbf{e}_1,..,\mathbf{e}_m\}]$

$\mathbf{Q}_b$:= **Select** $t_{R'\ell'_1}\bullet\mathbf{g}_q^{\ell'_1}$

From $\mathbf{R'}_{\ell'_1}\ t_{R'\ell'_1},....., \mathbf{R'}_{\ell'_p}\ t_{R'\ell'_p}$

where $t_{R'\ell'_{j-1}}\bullet\mathbf{f}_k^{\ell'_j} = t_{R'\ell'_j}\bullet\mathbf{g}_k^{\ell'_j}$, $1 \le k \le m_j$, $2 \le j \le p$ **and** $t_{R'\ell'_p}\bullet\mathbf{e}_i$= :$**a**•$\mathbf{h}_i$, $1 \le i \le m$

Tabela 5.1: Casos do Procedimento **GVA_1**

A seguir mostramos um exemplo desse caso de tradutor e em seguida mostramos que o tradutor gerado realiza a atualização solicitada na visão.

EXEMPLO 5.2:

Considere o esquema relacional apresentado na Figura 5.2, o esquema da visão Pedidos_v, apresentado na Figura 5.1 e as ACs de Pedidos_v definidas na seção 5.1.5.

Suponha que desejamos gerar o tradutor para inserções na tabela aninha **Listaltens** da visão **Pedidos_v**. Da AC de **Listaltens** (T_{pedido_v}•**listaltens** ≡ T_{pedido_rel}•fk_2^{-1}) temos que o caminho = fk_2^{-1}, onde a ligação multivalorada (fk_2^{-1}) é a última. Figura 5.11 mostra o tradutor para adições na coleção aninhada **Listaltens** de **Pedidos_v**, gerado pelo algoritmo.

De acordo com o tradutor gerado, um pedido de adição de um objeto (:new) do tipo T_{item} na coleção **listaltens** é traduzido em uma inserção da tupla (new.**codigo**, :new.**quantidade**, :new.**produto**, :parent.**codigo**) na tabela **Itens_rel**. Sempre que for realizada uma adição na coleção **listaltens** da visão **Pedidos_v**, o tradutor é disparado.

A seguir mostramos como foram geradas, pelo procedimento **GVA**$_1$, as subconsultas que computam os valores dos atributos **icodigo**, **iquantidade**, **iproduto** e **ipedido**.

- O atributo **icodigo** é um atributo monovalorado de valor atômico. Do Caso 2 do procedimento **GVA**$_1$ e da ACC ψ_{10}: [T_{item_v}•**codigo**] ≡ [T_{itens_rel}•**icodigo**], temos que $\mathbf{Q_{icodigo}}$ = :new.**codigo** ;

- O atributo **iquantidade** é um atributo monovalorado de valor atômico. Do Caso 2 do procedimento **GVA**$_1$ e da ACC ψ_{12}: [T_{item_v}•**quantidade**] ≡ [T_{itens_rel}•**iquantidade**], temos que $\mathbf{Q_{iquantidade}}$= **:new.quantidade;**

```
CREATE OR REPLACE TRIGGER AdicãoEmListaItens
INSTEAD OF INSERT ON NESTED TABLE listaItens OF Pedidos_v
BEGIN
 Insert Into Itens_rel

 ( icodigo, iquantidade, iproduto, ipedido )
  values ( :new.codigo,                          (de ψ10)
           :new.quantidade,                      (de ψ11)
            :new.produto,                        (de ψ12)
             :parent.código                      (da ACO ψ9)
          );

END;
```

Figura 5.11: Tradutor AdicãoEmListaItens

- O atributo **iproduto** é atributo monovalorado de valor atômico. Do Caso 2 do procedimento **GVA**$_1$ e da ACC ψ_{11}: [$\mathbf{T}_{item\_v}$•**produto**] ≡ [$\mathbf{T}_{itens\_rel}$•**iproduto**], temos que $\mathbf{Q}_{\mathbf{iproduto}}$=:new.**produto**;

- O atributo **ipedido** é parte da chave estrangeira FK2: Itens_rel[**ipedido**] ⊂ Pedidos_rel [**pcodigo**]. De ψ8:Tpedido_v.**listaItens** ≡ Tpedido_rel.fk_2^{-1} temos que o caminho $\varphi = fk_2^{-1}$. Do Caso 1 do procedimento **GVA**$_1$, da ACO ψ_9 [$\mathbf{T}_{item\_v}$,{**codigo**}] ≡ [$\mathbf{T}_{itens\_rel}$,{**icodigo**}], e dado que o caminho φ possui somente uma ligação, temos que: $\mathbf{Q}_{\mathbf{ipedido}}$= :parent.**código**.

Neste trabalho, consideramos que uma tradução é válida se esta realiza a atualização solicitada com o mínimo de efeitos colaterais [21][37]. Na seção A.1.1 do Anexo A, mostramos que o tradutor gerado realiza a atualização solicitada na visão. No Anexo A, seja:

- *U* a lista de atualizações requisitadas pelo tradutor (U= $\tau(\mu)$);
- σ_s e σ'_s são os estados de **S** imediatamente antes e depois de U;
- σ_v e σ'_v os estados da visão **V** em σ_s e σ'_s;
- σ''_v é o novo estado de **V** após a atualização solicitada μ..

Temos que provar que: $\sigma'_v = \sigma''_v$. (vide Figura 5.12).

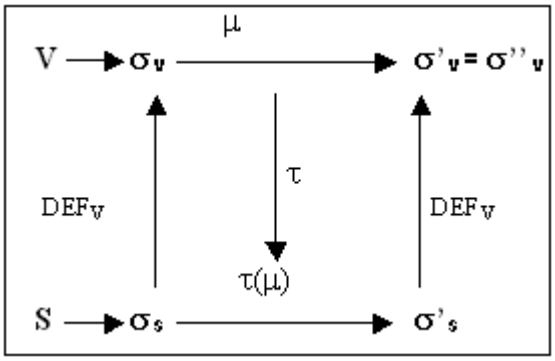


Figura 5.12: Tradução de Atualização de Visão

Caso 2: Ligação multivalorada não é a última

Neste caso, como mostrado em [37], para garantirmos que não existe ambigüidade no nível de dados, as seguintes condições devem ser satisfeitas:

1. A ACC de **lista_T$_c$** é dada por: [$\mathbf{T}_v$•**lista_T$_c$**] ≡ [$\mathbf{T}_{R_b}$•φ], onde:

1.1. $\varphi=\ell_1\bullet\ell_2\bullet..\bullet\ell_j\bullet..\bullet\ell_n$ é um caminho de $\mathbf{T}_{R_b}$, com ligações ℓ_1: $\mathbf{T}_{R_b}\rightarrow\mathbf{T}_{R\ell_1}$, l_i: $\mathbf{T}_{R\ell_{i-1}}\rightarrow\mathbf{T}_{R\ell_i}$, $2 \le i \le n$, como apresentado na Figura 5.13.

1.2. As ligações ℓ_1,., ℓ_{j-1} e ℓ_{j+1},.,ℓ_{n-1} e suas inversas são ligações monovaloradas, exceto a inversa da ligação ℓ_{j+1} que pode ser monovalorada ou multivalorada. Desta forma, temos que cada objeto de $\mathbf{R}_b$ está relacionado com um único objeto de $\mathbf{R}_{\ell_{j-1}}$ através do caminho $\ell_1\bullet\ell_2\bullet...\bullet\ \ell_{j-1}$ e vice-versa. Temos ainda que cada objeto de cada objeto de $\mathbf{R}_{\ell_j}$ está relacionado com um único objeto de $\mathbf{R}_{\ell_n}$ através do caminho $\ell_{j+1}\bullet..\bullet\ \ell_n$. A ligação ℓ_j é multivalorada virtual, obtida da inversa da ligação monovalorada ℓ: $\mathbf{T}_{R\ell_j}\rightarrow\mathbf{T}_{R\ell_{j-1}}$, onde $1< j < n$. Essa condição garante que a ligação multivalorada não é a última ligação do caminho φ.

1.3. $\mathbf{T}_{R\ell_j}$ é um tipo que representa um relacionamento N:M ou 1:N entre os tipos $\mathbf{T}_{R\ell_{j-1}}$ e $\mathbf{T}_{R\ell_{j+1}}$[44]. Assim sendo $\mathbf{T}_{R\ell_j}$ possui apenas duas ligações ℓ_{j+1}: $\mathbf{T}_{R\ell_j} \rightarrow \mathbf{T}_{R\ell_{j+1}}$ ($\mathbf{R}_{\ell_j}[\mathbf{f}_1{}^{\ell_{j+1}}, ..., \mathbf{f}_{m_{j+1}}{}^{\ell_{j+1}}]\subset \mathbf{R}_{\ell_{j+1}}[\mathbf{g}_1{}^{\ell_{j+1}},.., \mathbf{g}_{m_{j+1}}{}^{\ell_{j+1}}]$) e ℓ: $\mathbf{T}_{R\ell_j} \rightarrow \mathbf{T}_{R\ell_{j-1}}$ ($\mathbf{R}_{\ell_j}[\mathbf{g}_1{}^{\ell_j},.., \mathbf{g}_{m_j}{}^{\ell_j}] \subset \mathbf{R}_{\ell_{j-1}}[\mathbf{f}_1{}^{\ell_j}, ..., \mathbf{f}_{m_j}{}^{\ell_j}]$). Todos os outros atributos de $\mathbf{T}_{R\ell_j}$ admitem valor nulo.

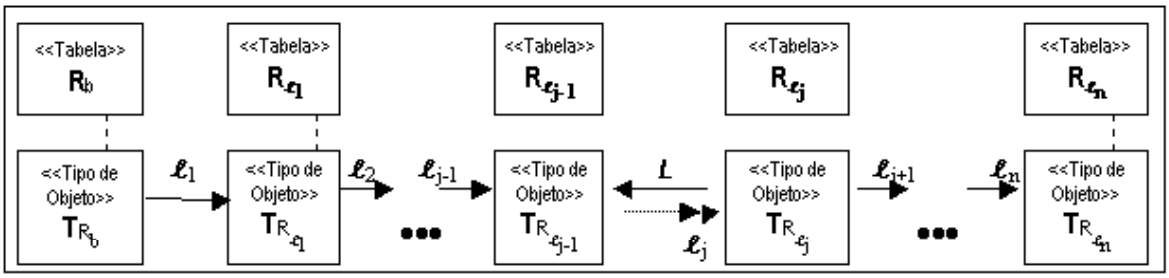


Figura 5.13: Caminho $\varphi=\ell_1\bullet\ell_2\bullet..\bullet\ell_{j-1}\bullet..\bullet\ell_n$ do tipo base $\mathbf{T_b}$

```
CREATE OR REPLACE TRIGGER AdicãoEm lista_Tc_Caso2
INSTEAD OF INSERT ON NESTED TABLE lista_Tc OF V
BEGIN
INSERT INTO Rℓj
(b1,...,bn)
VALUES (Qb1, ..., Qbn);
```

Figura 5.14: Tradutor para Adição Em lista_T_c(Caso 2)

Figura 5.14 mostra o molde do tradutor "AdicãoEm lista_T_c" para este caso. De acordo com o tradutor, um pedido de adição de um objeto (:new) do tipo $\mathbf{T_c}$ na coleção lista_T_c do objeto pai (:parent) é traduzido em uma inserção da tupla $\mathbf{t_{novo}} = (\mathbf{V_{b1}},... \mathbf{V_{bn}})$ na tabela $R_{\ell j}$, onde $\mathbf{b}_1, ...,\mathbf{b}_n$ são atributos das chaves estrangeiras na ligação ℓ_j ou ℓ_{j+1} .O valor do atributo $\mathbf{b}_i$ ($\mathbf{V_{bi}}$), $1 \leq i \leq n$, de $\mathbf{t_{novo}}$ é definido pela subconsulta $\mathbf{Q_{bi}}$, a qual computa o valor do atributo $\mathbf{b}_i$, $1 \leq i \leq n$, a partir do objeto inserido (:new) e seu pai (:parent), $2 \leq i \leq n$, e é definida com base nas ACs de $\mathbf{T_c}$ & $\mathbf{T}_{R_{\ell n}}$, e na ACC de lista_T_c ($\mathbf{T_v.lista\_T_c \equiv T_b.\varphi}$) de acordo com os casos definidos para o procedimento **GVA_2** (**G**era **V**alor de **A**tributo) na Tabela 5.2 abaixo:

Caso 1: **b** é um atributo da chave estrangeira na ligação ℓ_j do caminho $\varphi' = \ell_1 \bullet \ ... \ \bullet \ell_j$

Caso 1.1: **φ'** possui somente uma ligação $\boldsymbol{\ell_1}$ ($\boldsymbol{\varphi' = \ell_1}$) inversa da chave estrangeira dada por
$R_{\ell_1}[\mathbf{g}_1^{\ell_1}, ..., \mathbf{g}_{m_1}^{\ell_1}] \subset \mathbf{R_b}[\mathbf{f}_1^{\ell_1}, ..., \mathbf{f}_{m_1}^{\ell_1}]$ e $\mathbf{b} = \mathbf{f}_q^{\ell_1}$, $1 \leq q \leq m_1$, e
a ACO de $\mathbf{T}_{R_b}$ com $\mathbf{T_v}$ é dada por: $[\mathbf{T}_{R_b}, \{\mathbf{f}_1^{\ell_1}, ..., \mathbf{f}_{m_1}^{\ell_1}\}] \equiv [\mathbf{T_v}, \{\mathbf{d}_1,..,\mathbf{d}_{m_1}\}]$.

$\mathbf{Q_b}$:= :parent• $\mathbf{d_q}$

Caso 1.2: **φ'** possui mais de uma ligação (n>1) e $\mathbf{b} = \mathbf{g}_q^{\ell_j}$, $1 \leq q \leq m_j$, onde $\mathbf{g}_q^{\ell_j}$ é parte da chave estrangeira
$R_{\ell_{j-1}}[\mathbf{f}_1^{\ell_j}, ..., \mathbf{f}_{m_j}^{\ell_j}] \subset R_{\ell_j}[\mathbf{g}_1^{\ell_j},.., \mathbf{g}_{m_j}^{\ell_j}]$ e a ACO de $\mathbf{T}_{R_b}$ com $\mathbf{T_v}$ é dada por: $[\mathbf{T}_{R_b}, \{\mathbf{f}_1^{\ell_1}, ..., \mathbf{f}_{m_1}^{\ell_1}\}] \equiv [\mathbf{T_v}, \{\mathbf{d}_1,.., \mathbf{d}_{m_1}\}]$.

$\mathbf{Q_b}$:=**Select** $\mathbf{t}_{R_{\ell_{j-1}}} \bullet \mathbf{f}_q^{\ell_j}$
From $\mathbf{R}_{R_b}\ \mathbf{t}_{R_b}, \mathbf{R}_{\ell_1}\ \mathbf{t}_{R_{\ell_1}}, ..., \mathbf{R}_{\ell_{j-1}}\ \mathbf{t}_{R_{\ell_{j-1}}}$
Where $\mathbf{t}_{R_b} \bullet \mathbf{f}_q^{\ell_1} = \mathbf{t}_{R_{\ell_1}} \bullet \mathbf{g}_q^{\ell_1}$, $1 \leq q \leq m_1$ **and** $\mathbf{t}_{R_{\ell k-1}} \bullet \mathbf{f}_q^{\ell k} = \mathbf{t}_{R_{\ell k}} \bullet \mathbf{g}_q^{\ell k}$, $1 \leq q \leq m_k$, $2 \leq k \leq j-1$
and $\mathbf{t}_{R_b}.\mathbf{f}_1^{\ell_1}$ =:parent•$\mathbf{d_1}$ **and** ... **and** $\mathbf{t}_{R_b} \bullet \mathbf{f}_m^{\ell_1}$=:parent•$\mathbf{d}_{m_1}$

Caso 2: **b** é um atributo da chave estrangeira na ligação ℓ_{j+1} do caminho $\varphi'' = \ell_{j+1} \bullet ... \bullet \ell_n$

Caso 2.1: **φ''** possui somente uma ligação $\boldsymbol{\ell_n}$ ($\boldsymbol{\ell_j = \ell_n}$) inversa da chave estrangeira dada por
$\mathbf{R}_{\ell_{n-1}}[\mathbf{f}_1^{\ell_n}, ..., \mathbf{f}_{m_n}^{\ell_n}] \subset \mathbf{R}_{\ell_n}[\mathbf{g}_1^{\ell_n}, ..., \mathbf{g}_{m_n}^{\ell_n}]$ e $\mathbf{b} = \mathbf{f}_q^{\ell_n}$, $1 \leq q \leq m_n$, e
a ACO de $\mathbf{T}_{R_{\ell_n}}$ com $\mathbf{T_c}$ é dada por: $[\mathbf{T}_{R_{\ell_n}}, \{\mathbf{g}_1^{\ell_n}, ..., \mathbf{g}_{m_1}^{\ell_n}\}] \equiv [\mathbf{T_c}, \{\mathbf{d}_1,..,\mathbf{d}_{m_1}\}]$.

$\mathbf{Q_b}$:= :new• $\mathbf{d_q}$

Caso 2.2: **φ''** possui mais de uma ligação ($\boldsymbol{\ell_j \neq \ell_n}$) e $\mathbf{b} = \mathbf{f}_q^{\ell_{j+1}}$, $1 \leq q \leq m_{j+1}$, onde $\mathbf{f}_q^{\ell_{j+1}}$ é parte da chave estrangeira
$\mathbf{R}_{\ell_j}[\mathbf{f}_1^{\ell_{j+1}}, ..., \mathbf{f}_{m_{j+1}}^{\ell_{j+1}}] \subset \mathbf{R}_{\ell_{j+1}}[\mathbf{g}_1^{\ell_{j+1}},.., \mathbf{g}_{m_{j+1}}^{\ell_{j+1}}]$ e a ACO de $\mathbf{T}_{R_{\ell_n}}$ com $\mathbf{T_c}$ é dada por: $[\mathbf{T}_{R_{\ell_n}}, \{\mathbf{g}_1^{\ell_n}, ..., \mathbf{g}_{m_1}^{\ell_n}\}] \equiv [\mathbf{T_c}, \{\mathbf{d}_1,..,\mathbf{d}_{m_1}\}]$.

$\mathbf{Q_b}$:= **Select** $\mathbf{t}_{R_{\ell_{j+1}}} \bullet \mathbf{g}_q^{\ell_j}$
From $\mathbf{R}_{\ell_{j+1}}\ \mathbf{t}_{R_{\ell j+1}}, ..., \mathbf{R}_{\ell_n}\ \mathbf{t}_{R_{\ell_n}}$
Where $\mathbf{t}_{R_{\ell_{j+1}}} \bullet \mathbf{f}_q^{\ell_{j+2}} = \mathbf{t}_{R_{\ell_{j+2}}} \bullet \mathbf{g}_q^{\ell_{j+2}}$, $1 \leq q \leq m_j$ **and** $\mathbf{t}_{R_{\ell k-1}} \bullet \mathbf{f}_q^{\ell_k} = \mathbf{t}_{R_{\ell k}} \bullet \mathbf{g}_q^{\ell_k}$, $1 \leq q \leq m_k$, $j+2 \leq k \leq n$
and $\mathbf{t}_{R_{\ell_n}}.\mathbf{g}_1^{\ell_n}$ =:new•$\mathbf{d_1}$ **and** ... **and** $\mathbf{t}_{R_{\ell_n}} \bullet \mathbf{g}_m^{\ell_n}$=:new•$\mathbf{d}_{m_1}$

Tabela 5.2: Casos do Procedimento **GVA_2**

Na Seção A.1.2 do Anexo A mostramos que o tradutor gerado realiza a atualização solicitada na visão. A seguir mostramos um exemplo desse caso de tradutor.

EXEMPLO 5.2:

Considere, por exemplo, o esquema do banco de dados apresentado na Figura 5.15 (a), o esquema da visão Autores_v, apresentado na Figura 5.15 (b) e as ACs definidas abaixo. As ACS são geradas como definido na Seção 5.1.5.

Suponha que desejamos gerar o tradutor para inserções na tabela aninha **lista_pub** da visão **Autores_v**. Da AC de **lista_pub** ($[T_{autor\_v} \bullet \textbf{lista\_pub}] \equiv [T_{autor} \bullet fk1^{-1} \bullet fk2]$) temos que o caminho = $fk1^{-1} \bullet fk2$, onde a ligação multivalorada ($fk1^{-1}$) não é a última. Figura 5.16 mostra o tradutor para adições na coleção aninhada **lista_pub** de **Autores_v**, gerado pelo algoritmo.

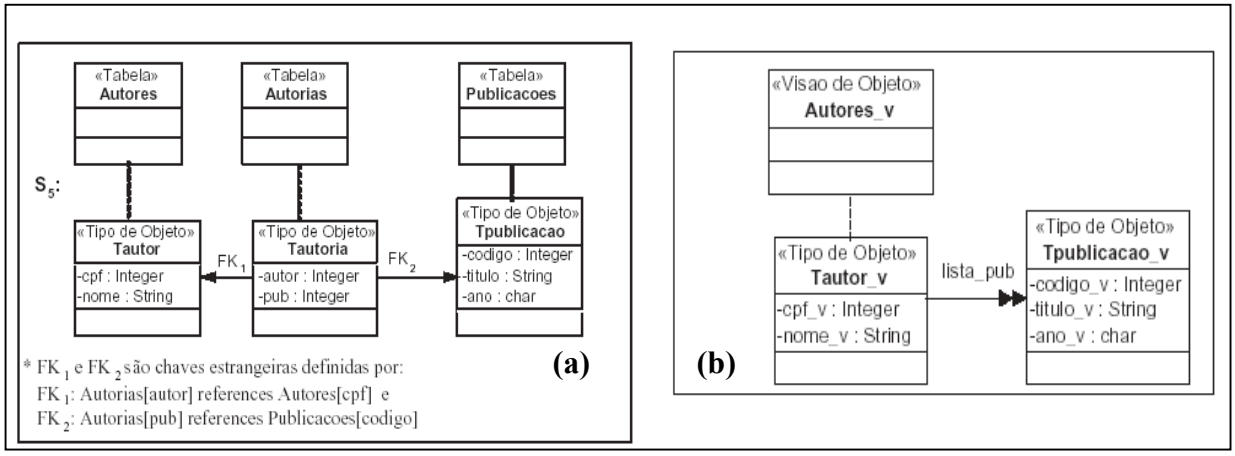


Figura 5.15: Esquema do Banco de Dados (a) e da Visão de Objetos Autores_v (b)

• Assertiva de Correspondência de Extensão
ψ_1: [Autores_v]≡[Autores]

• Assertivas de Correspondência de Objetos
ψ_2: $[T_{autor\_v}, \{cpf\_v\}] \equiv [T_{autor}, \{cpf\}]$
ψ_3: $[T_{publicacao\_v}, \{código\_v\}] \equiv [T_{publicacao}, \{codigo\}]$

• Assertivas de Correspondência de Caminhos entre os tipos T_{autor_v} e T_{autor}
ψ_4: $[T_{autor\_v} \bullet lista\_pub] \equiv [T_{autor} \bullet fk1^{-1} \bullet fk2]$
ψ_5: $[T_{autor\_v} \bullet cpf\_v] \equiv [T_{autor} \bullet cpf]$
ψ_6: $[T_{autor\_v} \bullet nome\_v] \equiv [T_{autor} \bullet nome]$

• Assertivas de Correspondência de Caminhos entre os tipos $T_{publicacao\_v}$ e $T_{publicacao}$
ψ_7: $[T_{publicacao\_v} \bullet codigo\_v] \equiv [T_{publicacao\_v} \bullet codigo]$

ψ_8: $[T_{publicacao\_v} \bullet titulo\_v] \equiv [T_{publicacao\_v} \bullet titulo]$
ψ_9: $[T_{publicacao\_v} \bullet ano\_v] \equiv [T_{publicacao\_v} \bullet ano]$

```
CREATE OR REPLACE TRIGGER AdicãoEmLlista_pub
INSTEAD OF INSERT ON NESTED TABLE lista_pub OF Autores_V
BEGIN
INSERT INTO Autorias
(autor,pub)
 values (:parent.cpf_v, :new.codigo_v);
```

Figura 5.16: Tradutor AdicãoEmLlista_pub

De acordo com o tradutor gerado, um pedido de adição de um objeto (:new) do tipo $T_{publicacao\_v}$ na coleção **lista_pub** é traduzido em uma inserção da tupla (:parent.cpf_v, :new.codigo_v) na tabela **Autorias**. Sempre que for realizada uma adição na coleção **lista_pub** da visão **Autores_v**, o tradutor é disparado.

A seguir mostramos como foram geradas, pelo procedimento $\mathbf{GVA}_2$, as subconsultas que computam os valores dos atributos **autor** e **pub**. Dado que $fk1^{-1}$ é a ligação multivalorada no caminho $\varphi = fk1^{-1} \bullet fk2$. Logo temos que : $\varphi' = FK1^{-1}$ e $\varphi'' = FK2$.

- O atributo **autor** é um atributo da chave estrangeira Fk1:Autorias[**autor**]⊂ Autores[**cpf**] no caminho $\varphi' = FK1^{-1}$. Do Caso 1.1 do procedimento $\mathbf{GVA}_2$ e da ACO $\psi 2$: $[T_{autor\_v}, \{\mathbf{cpf\_v}\}] \equiv [T_{autor}, \{\mathbf{cpf}\}]$, dado que o caminho φ' possui somente uma ligação($\varphi' = FK1^{-1}$), temos que $\mathbf{Q_{autor}}$=:parent.**cpf_v**.
- O atributo **pub** é um atributo da chave estrangeira FK2: Autorias[**pub**] ⊂ Publicacao[**codigo**] no caminho $\varphi'' = FK2$. Do Caso 2.1 do procedimento $\mathbf{GVA}_2$ e da ACO $\psi 3$: $[T_{publicacao\_v}, \{\mathbf{codigo\_v}\}] \equiv [T_{publicacao}, \{\mathbf{codigo}\}]$, temos que: $\mathbf{Q_{pub}}$= :**new.codigo_v**.

5.4 Definição de Tradutores para Operações de Remoção em Coleções Aninhadas

Nesta seção apresentamos o algoritmo para geração dos tradutores (*instead of triggers*) para a operação de remoção em coleções aninhadas. No Oracle9i o comando DELETE pode ser usado para remover objetos em coleções aninhadas. Por exemplo, considere a visão **Pedidos_v** apresentada na Figura 5.5. Para remover um item na coleção aninhada **listaItens** do pedido da visão **Pedidos_v** da Figura 5.5, cujo código é 1, podemos usar a atualização mostrada na Figura 5.17. No Oracle 9i, **:old** refere-se ao objeto removido

na coleção aninhada, e **:parent** refere-se ao objeto "pai" do objeto removido, que no exemplo é o pedido de código=1 da visão **Pedidos_v**. Note que a subconsulta "TABLE(SELECT v•**listaltens** FROM **Pedidos_v** v WHERE v•**codigo**=1)" retorna a coleção aninhada do objeto pai selecionado. A condição na cláusula WHERE da subconsulta deve garantir que um único objeto é selecionado.

```
DELETE FROM TABLE( SELECT v•listaltens FROM Pedidos_v v WHERE v•codigo =1) v1
WHERE v1.codigo=5;
```

Figura 5.17: Exemplo de remoção na coleção aninhada **listaltens** da visão **Pedidos_v**

No resto desta seção, considere a visão **V** cujos objetos são do tipo $\mathbf{T_v}$ e **lista_T**$_c$ um atributo multivalorado (coleção aninhada) de $\mathbf{T_v}$, cujo valor é um conjunto de objetos do tipo $\mathbf{T_c}$, como mostrado na Figura 5.8 (a). Suponha que: (i) a extensão da visão **V** é definida por uma ACE de equivalência $\mathbf{V \equiv R_b}$ ou de subconjunto $\mathbf{V \subset R_b}$, onde $\mathbf{R_b}$ é a tabela base cujos objetos são do tipo $\mathbf{T_b}$ e (ii) A ACC de **lista_T**$_c$ é dada por: $\mathbf{T_v \bullet lista\_T_c \equiv T_{R_b} \bullet \varphi}$, onde $\varphi = \ell_1 \bullet \ell_2 \bullet .. \bullet \ell_n$ é um caminho de $\mathbf{T_{R_b}}$ (Figura 5.18), com ligações ℓ_1: $\mathbf{T_{R_b} \rightarrow T_{R_{\ell 1}}}$, l_i: $\mathbf{T_{R_{\ell i-1}} \rightarrow T_{R_{\ell i}}}$, $2 \leq i \leq n$ e ℓ_j é uma ligação multivalorada.

No caso em que o esquema do banco de dados é relacional, no caminho $\varphi = \ell_1 \bullet \ell_2 \bullet \ldots \bullet \ell_{j-1} \bullet ... \bullet \ell_n$, temos que $\ell_1, .. \ell_{j-1} .., e \ \ell_n$ são ligações de chave estrangeira ou inversas de chave estrangeira. No resto desta seção, suponha que ℓ_1 é uma ligação de chave estrangeira dada por $\mathbf{R_b}\ [\mathbf{f_1}^{\ell 1}, ..., \mathbf{f_{m_1}}^{\ell 1}] \subset \mathbf{R_{\ell 1}}\ [\mathbf{g_1}^{\ell 1}, .., \mathbf{g_{m_1}}^{\ell 1}]$ ou inversa da chave estrangeira dada por $\mathbf{R_{\ell 1}}$ $[\mathbf{g_1}^{\ell 1}, ..., \mathbf{g_{m_1}}^{\ell 1}] \subset \mathbf{R_b}\ [\mathbf{f_1}^{\ell 1}, ..., \mathbf{f_{m_1}}^{\ell 1}]$; e ℓ_i é uma ligação de chave estrangeira dada por $\mathbf{R_{\ell_{i-1}}}$ $[\mathbf{f_1}^{\ell_i}, ..., \mathbf{f_{m_i}}^{\ell_i}] \subset \mathbf{R_{\ell_i}}\ [\mathbf{g_1}^{\ell_i}, .., \mathbf{g_{m_i}}^{\ell_i}]$ ou inversa da chave estrangeira dada por $\mathbf{R_{\ell_i}}\ [\mathbf{g_1}^{\ell_i}, ..., \mathbf{g_{m_i}}^{\ell_i}] \subset$ $\mathbf{R_{\ell_{i-1}}}\ [\mathbf{f_1}^{\ell_i}, ..., \mathbf{f_{m_i}}^{\ell_i}]$, $2 \leq i \leq n$.

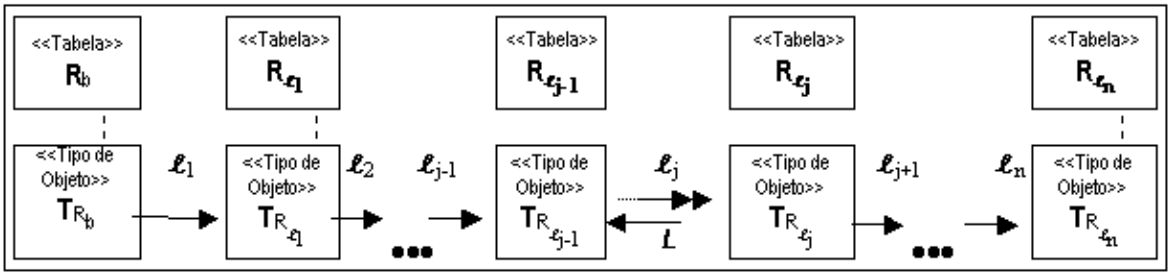


Figura 5.18: Caminho $\varphi=\ell_1\bullet\ell_2\bullet..\bullet\ell_{j-1}\bullet..\bullet\ell_n$ do tipo base $\mathbf{T}_{R_b}$

O algoritmo gera tradutores para os dois casos descritos a seguir. Em ambos os casos, só poderá existir uma única ligação multivalorada no caminho de derivação de lista_T_c (φ), uma vez que na existência de mais de uma ligação ocorrerá ambigüidade a nível de dados. O Caso 1 trata a situação em que a ligação multivalorada é a ultima ligação de φ, e ,o Caso 2, a situação em que a ligação multivalorada não é a ultima. Mostraremos que nestes casos não existe ambigüidade a nível de dados e a tradução pode ser gerada em tempo de projeto.

Caso 1: Ligação multivalorada é a última

Neste caso, como mostrado em [37], para garantirmos que não existe ambigüidade no nível de dados, as seguintes condições devem ser satisfeitas:

A ACC de **lista_T_c** é dada por [$\mathbf{T_v}\bullet$**lista_T_c**] $\equiv$[$\mathbf{T}_{R_b}\bullet\varphi$], onde:

1. $\varphi=\ell_1\bullet\ell_2\bullet..\bullet\ell_j\bullet..\bullet\ell_n$ é um caminho de $\mathbf{T}_{R_b}$, com ligações ℓ_1: $\mathbf{T}_{R_b}\rightarrow\mathbf{T}_{R_{\ell 1}}$, l_i: $\mathbf{T}_{R_{\ell i-1}}\rightarrow\mathbf{T}_{R_{\ell i}}$, $2 \leq i \leq n$, como apresentado na Figura 5.18;
2. ℓ_j é a ligação multivalorada, tal que: ℓ_j é a última ligação do caminho φ (j=n), então as ligações $\ell_1,..,\ell_{n-1}$ e suas inversas são monovaloradas, ou seja, cada objeto de $\mathbf{R_b}$ está relacionado com um único objeto de $\mathbf{R}_{\ell_{n-1}}$ através do caminho $\ell_1\bullet..\bullet\ell_{n-1}$ e vice-versa.

A Figura 5.19 mostra o molde do tradutor "RemoçãoEm lista_T_c_Caso1" para este caso. De acordo com o tradutor, um pedido de remoção de um objeto (:old) do tipo $\mathbf{T_c}$ na coleção lista_T_c do objeto pai (:parent) é traduzido em uma remoção de uma tupla $\mathbf{t}_{R_{\ell_n}}$ da tabela R_{ℓ_n}, tal que :old $\equiv \mathbf{t}_{R_{\ell_n}}$. Note que, de acordo com a ACO de $\mathbf{T_c}$ & $\mathbf{T}_{R_{\ell_n}}$ ψ :[$\mathbf{T_c}$,{$\mathbf{f}_1,\mathbf{f}_2,...,\mathbf{f}_w$}] $\equiv$ [$\mathbf{T}_{R_{\ell_n}}$ {$\mathbf{g}_1,\mathbf{g}_2,...,\mathbf{g}_w$}], :old $\equiv \mathbf{t}_{R_{\ell_n}}$ sss "$\mathbf{t}_{R_{\ell_n}}\bullet\mathbf{f}_k$ =:old$\bullet\mathbf{g}_k$, $1\leq k \leq w$. Na Seção A.2.1 do Anexo A mostramos que o tradutor gerado realiza a atualização solicitada na visão. A seguir mostramos um exemplo desse caso de tradutor.

```
CREATE OR REPLACE TRIGGER RemoçãoEm lista_Tc_Caso1
INSTEAD OF DELETE ON NESTED TABLE lista_Tc OF V
BEGIN

DELETE FROM R_ℓn t_Rℓn
WHERE  t_Rℓn• f1 =:old• g1 .. AND t_Rℓn• fw =:old•cw;

END.
```

Figura 5.19: Tradutor para "RemoçãoEm lista_T_c_Caso1"

EXEMPLO 5.3:

Considere o esquema do banco de dados apresentado na Figura 5.2, o esquema da visão **Pedidos_v**, apresentado na Figura 5.1 e as ACs de **V** definidas na seção 5.1.5.

Suponha que desejamos gerar o tradutor para remoções na tabela aninha **listaItens** da visão **Pedidos_v**. Da AC de **listaItens** ($T_{pedido\_v.listaItens} \equiv T_{pedido\_rel}.fk_2^{-1}$) temos que o caminho = fk_2^{-1}, onde a ligação multivalorada (fk_2^{-1}) é a última. Figura 5.20 mostra o tradutor para remoções na coleção aninhada **listaItens** de **Pedidos_v**, gerado pelo algoritmo.

De acordo com o tradutor gerado, um pedido de remoção de um objeto (:old) do tipo $\mathbf{T}_{item}$ na coleção **listaItens** é traduzido em uma remoção de uma tupla $\mathbf{t}_{\mathbf{itens\_rel}}$ da tabela **Itens_rel**, tal que :old **≡** $\mathbf{t}_{\mathbf{itens\_rel}}$. Note que, de acordo com a ACO de $\mathbf{T}_{item}$ & $\mathbf{T}_{itens\_rel}$ $[\mathbf{T}_{item\_v},\{codigo\}] \equiv [\mathbf{T}_{itens\_rel},\{icodigo\}]$, :old **≡** $\mathbf{t}_{\mathbf{itens\_rel}}$ sss "$\mathbf{t}_{itens\_rel}$•**icodigo**=:old•**codigo**". Sempre que for realizada uma remoção na coleção **listaItens** da visão **Pedidos_v**, este tradutor é disparado.

```
CREATE OR REPLACE TRIGGER RemoçãoEmListaItens
INSTEAD OF DELETE ON NESTED TABLE listaItens OF Pedidos_v
BEGIN
 Delete from Itens_rel t_itens_rel where t_itens_rel•icodigo=:old•codigo;
END;
```

Figura 5.20: Tradutor "RemoçãoEmListaItens"

Caso2: Ligação multivalorada não é a última

Neste caso, como mostrado em [37], para garantirmos que não existe ambigüidade no nível de dados, as seguintes condições devem ser satisfeitas:

A ACC de **lista_Tc** é dada por $[\mathbf{T_v}\bullet\mathbf{lista\_T_c}] \equiv [\mathbf{T}_{R_b}\bullet\varphi]$, onde:

1. $\varphi=\ell_1\bullet\ell_2\bullet..\bullet\ell_j\bullet..\bullet\ell_n$ é um caminho de $\mathbf{T}_{R_b}$, com ligações ℓ_1: $\mathbf{T}_{R_b} \rightarrow \mathbf{T}_{R_{\ell 1}}$, l_i: $\mathbf{T}_{R_{\ell i-1}} \rightarrow \mathbf{T}_{R_{\ell i}}$, $2 \leq i \leq n$, como apresentado na Figura 5.18;

2. ℓ_j não é a última ligação do caminho φ (j<n), então as ligações ℓ_1,.. ℓ_{j-1} e ℓ_{j+1}, ..., ℓ_n e suas inversas são monovaloradas. Desta forma, temos que cada objeto de $\mathbf{R_b}$ está relacionado com um único objeto de $\mathbf{R}_{\ell_{j-1}}$ através do caminho $\ell_1 \bullet \ell_2 \bullet$ ... $\bullet \ell_{j-1}$ e vice-versa. Temos ainda que cada objeto de $\mathbf{R}_{\ell_j}$ está relacionado com um único objeto de $\mathbf{R}_{\ell_n}$ através do caminho $\ell_{j+1} \bullet$ ... $\bullet \ell_n$.

A Figura 5.21 mostra o molde do tradutor "RemoçãoEm lista_T_c_Caso2" para este caso. De acordo com o tradutor, um pedido de remoção de um objeto (:old) do tipo $\mathbf{T_c}$ na coleção **lista_T_c** do objeto pai (:parent) é traduzido em uma remoção de uma tupla $\mathbf{t}_{R_{\ell_j}}$ da tabela R_{ℓ_j}, tal que $\mathbf{t}_{R_{\ell_j}} \bullet \ell_{j+1} \bullet .. \bullet \ell_n = \mathbf{t}_{R_{\ell_n}}$,onde $\mathbf{t}_{R_{\ell_n}} \equiv$:old, e $\mathbf{t}_{Rb} \bullet \ell_1 \bullet .. \bullet \ell_j = \mathbf{t}_{R_{\ell_j}}$,onde $\mathbf{t}_{Rb} \equiv$:parent. Na seção A.2.2 do Anexo A mostramos que o tradutor gerado realiza a atualização solicitada na visão.

```
CREATE OR REPLACE TRIGGER RemoçãoEm lista_Tc_Caso1
INSTEAD OF DELETE ON NESTED TABLE lista_Tc OF V
BEGIN
DELETE FROM R_ℓj t_Rℓj
WHERE t_Rℓj•f_s^ℓj+1 =(Select t_Rℓj+1•g_s^ℓj+1
                  From R_ℓj+1 t_Rℓj+1 ,....., R_ℓn t_Rℓn
                  Where t_Rℓj+1•f_q^ℓj+2= t_Rℓj+2•g_q^ℓj+2 , 1≤ k≤ m_j+2 AND t_Rℓp-1•f_q^ℓp= t_Rℓp•g_q^ℓp ,
                        1≤ q≤ m_p , j+1≤p≤n  AND t_Rℓn•f_q^ln = :old•d_q ) , 1≤ s ≤ m_j+1
            AND
        t_Rℓj•g_s^ℓj =(Select t_Rℓj-1•f_s^ℓj
                  From R_Rb t_Rb, R_ℓ1 t_Rℓ1,....., R_ℓj-1 t_Rℓj-1
                  Where t_Rb•f_q^ℓ1= t_Rℓ1•g_q^ℓ1, 1≤ q≤m_1 and t_Rℓk-1•f_q^ℓk=t_Rℓk•g_q^ℓk, 1≤q≤m_k ,
                        2≤k≤j-1and t_Rb.f_1^ℓ1 =:parent•d_1 and ... and
                        t_Rb•f_m^ℓ1=:parent•d_mq ) , 1≤ s ≤ m_j;
END.
```

Figura 5.21: Tradutor para "RemoçãoEm lista_T_c_Caso2"

5.5 Definição de Tradutores para Operações de Adição em Visões

Nesta seção apresentamos o algoritmo para geração dos tradutores para a operação de adição em visões de objetos. Para adicionar um pedido na visão **Pedidos_v** da Figura 5.5,

podemos usar a atualização mostrada na Figura 5.22. No Oracle 9i,**:new** refere-se ao objeto inserido na visão **Pedidos_v**.

```
INSERT INTO Pedidos_v
(codigo,data, dataEntrega, endereçoEntrega, cliente_ref, listaItens)
 VALUES (1,'11/09/2004','23/09/2004',
            Tendereço_v('rua x',tabuleiro','ceara','60960000',
            Select REF(c) From Clientes_v Where c.código='1',
             TlistaItens(Titem(1,2,10),Titem(2,3,20));
```

Figura 5.22: Exemplo de inserção na visão **Pedidos_v**

No resto desta seção, considere a visão **V** cujos objetos são do tipo T_v, como mostrado na Figura 5.23. Suponha que: (i) a extensão da visão **V** é definida por uma ACE de equivalência $V \equiv R_b$ ou de subconjunto $V \subset R_b$, onde R_b é a tabela base cujos objetos são do tipo T_b e (ii) $[T_v,\{c_1,c_2,...,c_w\}] \equiv [T_{R_b}\{d_1,d_2,...,d_w\}]$ é a ACO de T_v & T_{R_b}.

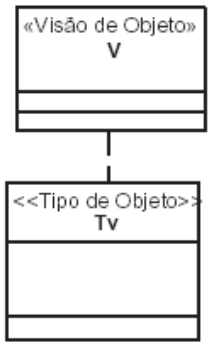


Figura 5.23: Esquema da Visão **V**

O algoritmo gera tradutores para inserção na visão **V**. Como mostrado em [37], para garantirmos que não existe ambigüidade a nível de dados, as seguintes condições devem ser satisfeitas:

1. As assertivas de correspondência dos atributos multivalorados de T_v devem ser de um dos tipos definidos abaixo:
   1.1. $[T_v \bullet a] \equiv [T_{R_b} \bullet \varphi]$, onde **a** é uma coleção aninhada de T_v e $\varphi = \ell_1 \bullet \ell_2 ... \bullet \ell_n$, como definido na seção 5.3;
   1.2. $[T_v \bullet a] \equiv [T_{R_b}\{b_1,b_2, ..., b_n\}]$, onde **a** é um atributo multivalorado de valor atômico e $b_1,b_2, ..., b_n$ atributos de valor atômico de T_{R_b};

2. As assertivas de correspondência de caminhos monovalorados entre $\mathbf{T}_v$ e $\mathbf{T}_{R_b}$ podem ser de um dos tipos definidos abaixo:

   2.1. $[\mathbf{T}_v \bullet \mathbf{a}] \equiv [\mathbf{T}_{R_b} \bullet \mathbf{b}]$, onde **a** é um atributo monovalorado de $\mathbf{T}_v$ e **b** é um atributo de valor atômico de $\mathbf{T}_{R_b}$;

   2.2. $[\mathbf{T}_v \bullet \mathbf{a}, \{\mathbf{a}_1, \mathbf{a}_2, ..., \mathbf{a}_n\}] \equiv [\mathbf{T}_{R_b} \{\mathbf{b}_1, \mathbf{b}_2, ..., \mathbf{b}_n\}]$, onde **a** um atributo de $\mathbf{T}_v$, cujo tipo $\mathbf{T}_a$ contém atributos atômicos $\mathbf{a}_1, \mathbf{a}_2, .., \mathbf{a}_n$ e $\mathbf{b}_1, \mathbf{b}_2, .., \mathbf{b}_n$ atributos de valor atômico de $\mathbf{T}_{R_b}$;

Para os casos 2.3 e 2.4 abaixo, considere $\varphi = \ell_1 \bullet \ell_2 ... \bullet \ell_p$ um caminho de $\mathbf{T}_{R_b}$, onde $\boldsymbol{\ell_1}$ é uma chave estrangeira de R_b dada por $R_b\ [\mathbf{f}_1^{\ell 1}, ..., \mathbf{f}_{m_1}^{\ell 1}] \subset R_{\ell_1}\ [\mathbf{g}_1^{\ell 1}, .., \mathbf{g}_{m_1}^{\ell 1}]$, e ℓ_i, é uma ligação de chave estrangeira dada por $\mathbf{R}_{\ell_{i-1}}\ [\mathbf{f}_1^{\ell i}, ..., \mathbf{f}_{m_i}^{\ell i}] \subset \mathbf{R}_{\ell_i}\ [\mathbf{g}_1^{\ell i}, .., \mathbf{g}_{m_i}^{\ell i}]$ ou inversa da chave estrangeira dada por $\mathbf{R}_{\ell_i}\ [\mathbf{g}_1^{\ell i}, ..., \mathbf{g}_{m_i}^{\ell i}] \subset \mathbf{R}_{\ell_{i-1}}\ [\mathbf{f}_1^{\ell i}, ..., \mathbf{f}_{m_i}^{\ell i}]$, $2 \leq i \leq p$, tal que: ℓ_1 é uma ligação monovalorada direta cuja inversa pode ser multivalorada ou monovalorada. As ligações $\ell_2 . \bullet .. \bullet \ell_p$ e suas inversas são monovaloradas. Desta forma, temos que cada objeto de $\mathbf{R}_{\ell_1}$ está relacionado com um único objeto de $\mathbf{R}_{\ell_p}$ e vice-versa;

   2.3. $[\mathbf{T}_v \bullet \mathbf{a}] \equiv [\mathbf{T}_{R_b} \bullet \varphi \bullet \mathbf{k}]$, onde **a** é um atributo de valor atômico de $\mathbf{T}_v$ e **k** é um atributo e identificador de $\mathbf{R}_{\ell_p}$. Esta condição garante que uma seleção em $\mathbf{R}_{\ell_p}$, dado um valor de **k**, retorna uma única instância de $\mathbf{R}_{\ell_p}$;

   2.4. $[\mathbf{T}_v \bullet \mathbf{a}] \equiv [\mathbf{T}_{R_b} \bullet \varphi]$, onde **a** é um atributo de valor estruturado ou de referência do tipo $\mathbf{T}_a$.

A Figura 5.24 mostra o molde do tradutor "AdicãoNaVisaoV" para adição na visão **V**. De acordo com o tradutor, um pedido de adição de um objeto (:new) do tipo $\mathbf{T}_v$ na visão **V** é traduzido em uma atualização no banco de dados em dois passos. No passo1 , é efetuada uma inserção da tupla $\mathbf{t}_{novo} = (\mathbf{V}_{b1}, ... \mathbf{V}_{bn})$ na tabela $\mathbf{R_b}$. O valor do atributo $\mathbf{b}_i$ ($\mathbf{V}_{bi}$), $1 \leq i \leq n$, de $\mathbf{t}_{novo}$ é definido pela subconsulta $\mathbf{Q}_{bi}$, a qual computa o valor do atributo $\mathbf{b}_i$, $1 \leq i \leq n$, a partir do objeto inserido (:new), e é definida com base nas ACs de $\mathbf{T}_v$ & $\mathbf{T}_{Rb}$, de acordo com os casos definidos do procedimento $\mathbf{GVA}_3$ (**G**era **V**alor de **A**tributo) na Tabela 5.3. Neste passo, são atribuídos valores somente aos atributos monovalorados e multivaorados, cuja AC é da forma $[\mathbf{T}_v \bullet \mathbf{a}] \equiv [\mathbf{T}_{R_b} \{\mathbf{b}_1, \mathbf{b}_2, ..., \mathbf{b}_n\}]$, do objeto :new. Os outros tipos de atributos multivalorados do objeto :new são tratados no Passo 2.

```
CREATE OR REPLACE TRIGGER AdicãoNaVisaoV
INSTEAD OF INSERT ON V
[DECLARE ]
BEGIN
Lista_Tc _nt := :new.lista_Tc
/*Passo 1 */
INSERT INTO Rb
( b1, ..., bn)
 values (Qb1, ..., Qbn);
/* Passo 2 - Para cada atributo multivalorado Lista_Tc em V faça: */
FOR I in.. Lista_Tc _nt.count LOOP
   INSERT into TABLE(Select  v.lista_tc FROM V v Where v. c1 = :new.c1 ..... AND v.
cw = :new.cw)
  VALUES(Lista_Tc _nt(i);
END Loop;END;
```

Figura 5.24: Template do tradutor "AdiçãoNaVisãoV"

No Passo 2, para cada atributo multivalorado **Lista-T_c** do objeto :new é solicitado a adição dos objetos na coleção aninhada **Lista-T_c** do objeto inserido :new, isto é, do objeto da visão **V**. Note que o pedido de atualização na coleção aninhada é traduzido em atualizações no banco de dados usando o tradutor para adição na tabela aninhada **Lista-T_c** (AdiçãoEmLista_T_c) da visão V, o qual é gerado como discutido na Seção 5.3. Na Seção A.3 do Anexo A mostramos que o tradutor gerado realiza a atualização solicitada na visão. A seguir mostramos um exemplo desse caso de tradutor.

Caso 1: **b** monovalorado de valor atômico e ψ:[T_v•**a**] ≡ [T_{Rb}•**b**], onde **a** é um atributo monovalorado de T_v

Q_b:=:new•**a** ;

Caso 2: **b=di**, **di** é monovalorado de valor atômico e ψ: [T_v•**a**] ≡[$T_{Rb,}$ {d_1,d_2,...,d_w}], onde **a** é um atributo multivalorado de T_v do tipo varray

Q_b :=:new•**a[i]** ;

Caso 3: **b** = **di**, **di** é monovalorado de valor atômico e ψ:[T_v•**a**,{a_1,a_2,...,a_w}]≡[T_{Rb},{d_1,d_2,..., d_w}]

Q_b :=:new•**a**•a_i;

Para os casos 4 e 5 considere $\varphi = \ell_1 \bullet \ell_2 ... \bullet \ell_p$ um caminho de $T_{R\ell_b}$, onde ℓ_1 é uma chave estrangeira de R_b dada por $R_b[\mathbf{f}_1^{\ell_1}, ..., \mathbf{f}_{m_1}^{\ell_1}] \subset R_{\ell_1}[\mathbf{g}_1^{\ell_1},.., \mathbf{g}_{m_1}^{\ell_1}]$, e ℓ_i, é uma ligação de chave estrangeira dada por $\mathbf{R}_{\ell_{i-1}}[\mathbf{f}_1^{\ell_i}, ..., \mathbf{f}_{m_i}^{\ell_i}] \subset \mathbf{R}_{\ell_i}[\mathbf{g}_1^{\ell_i},.., \mathbf{g}_{m_i}^{\ell_i}]$ ou inversa da chave estrangeira dada por $\mathbf{R}_{\ell_i}[\mathbf{g}_1^{\ell_i}, ..., \mathbf{g}_{m_i}^{\ell_i}] \subset \mathbf{R}_{\ell_{i-1}}[\mathbf{f}_1^{\ell_i}, ..., \mathbf{f}_{m_i}^{\ell_i}]$, $2 \le i \le p$;.

Caso 4: **b** é um atributo da ligação ℓ_1 e [T_v•**a**]≡ [T_{Rb}•φ•**k**] e **b** = $\mathbf{f}_q^{\ell_1}$, $1 \le q \le m_1$.

Q_b:= **Select** $t_{R\ell_1} \bullet \mathbf{g}_q^{\ell_1}$

From $\mathbf{R}_{\ell_1}$ $t_{R\ell_1}$,….., $\mathbf{R}_{\ell_p}$ $t_{R\ell_p}$

Where $t_{R\ell_{j-1}} \bullet \mathbf{f}_k^{\ell_j} = t_{R\ell_j} \bullet \mathbf{g}_k^{\ell_j}$, $1 \le k \le m_j$, $2 \le j \le p$ **and** $t_{R\ell_p} \bullet$**k**= :**new**•**a**

Caso 5: **b** é um atributo da ligação ℓ_1 e [T_v•**a**] ≡[T_{Rb}•φ].

Caso 5.1: **a** é um atributo de valor estruturado do tipo T_a

Caso 5.1.1: φ possui somente uma ligação ℓ_1 ($[T_v \bullet \mathbf{a}] \equiv [T_{Rb} \bullet \ell_1]$), $\mathbf{b} = \mathbf{f}_q^{\ell 1}$, $1 \le q \le m_1$, e ACO de $T_{R\ell_1}$ com T_a é dada por: $[T_a, \{\mathbf{h}_1, .., \mathbf{h}_{m_1}\}] \equiv [T_{R\ell'1}, \{\mathbf{g}_1^{\ell 1}, .., \mathbf{g}_{m_1}^{\ell 1}\}]$

Q_b:= :new•a•$\mathbf{h_q}$

Caso 5.1.2: φ possui mais que uma ligação (p>1) e $\mathbf{b} = \mathbf{f}_q^{\ell 1}$, $1 \le q \le m_1$ e ACO de $T_{R\ell_p}$ com T_a é dada por: $[T_a, \{\mathbf{h}_1, .., \mathbf{h}_m\}] \equiv [T_{R\ell_p} \{\mathbf{e}_1, .., \mathbf{e}_m\}]$

Q_b:= **Select** $t_{R'\ell_1} \bullet \mathbf{g}_q^{\ell 1}$

From $\mathbf{R}_{\ell 1}$ $t_{R\ell 1}$,….., $\mathbf{R}_{\ell_p}$ $t_{R\ell_p}$

where $t_{R\ell_{j-1}} \bullet \mathbf{f}_k^{\ell_j} = t_{R\ell_j} \bullet \mathbf{g}_k^{\ell_j}$, $1 \le k \le m_j$, $2 \le j \le p$ **and** $t_{R\ell_p} \bullet \mathbf{e}_i$= :**new•a•h**$_i$>, $1 \le i \le m$

Caso 5.2: **a** é um atributo de referência do tipo T_a

/* Seja $**a** o objeto referenciado por :new.**a**. Nesse caso é realizada a chamada da função UTL_REF_SELECT OBJETC($**a**, :new.**a**) no tradutor gerado. UTL_REF_SELECT OBJETCT é uma função do Oracle utilizada para "desreferenciar" um objeto, então temos que a variável $**a** do tipo $\mathbf{T}_a$ recebe o objeto da referência "**:new.a**";*/

Caso 5.2.1: φ possui somente uma ligação ℓ_1 ($[T_v \bullet \mathbf{a}] \equiv [T_{Rb} \bullet \ell_1]$), $\mathbf{b} = \mathbf{f}_q^{\ell 1}$, $1 \le q \le m_1$, e ACO de $T_{R\ell_1}$ com T_a é dada por: $[T_a, \{\mathbf{h}_1, .., \mathbf{h}_{m_1}\}] \equiv [T_{R\ell'1}, \{\mathbf{g}_1^{\ell 1}, .., \mathbf{g}_{m_1}^{\ell 1}\}]$

Q_b:= :$**a**•$\mathbf{h_q}$

Caso 5.2.2: φ possui mais que uma ligação (p>1) e $\mathbf{b} = \mathbf{f}_q^{\ell 1}$, $1 \le q \le m_1$ e ACO de $T_{R\ell_p}$ com T_a é dada por: $[T_a, \{\mathbf{h}_1, .., \mathbf{h}_m\}] \equiv [T_{R\ell_p} \{\mathbf{e}_1, .., \mathbf{e}_m\}]$

Q_b:= **Select** $t_{R\ell_1} \bullet \mathbf{g}_q^{\ell 1}$

From $\mathbf{R}_{\ell 1}$ $t_{R\ell 1}$,….., $\mathbf{R}_{\ell_p}$ $t_{R\ell_p}$

Where $t_{R\ell_{j-1}} \bullet \mathbf{f}_k^{\ell_j} = t_{R\ell_j} \bullet \mathbf{g}_k^{\ell_j}$, $1 \le k \le m_j$, $2 \le j \le p$ **and** $t_{R\ell_p} \bullet \mathbf{e}_i$= :**$a•h**$_i$, $1 \le i \le m$

Tabela 5.3: Casos do Procedimento **GVA$_3$**

EXEMPLO 5.7:

Considere o esquema relacional apresentado na Figura 5.2, o esquema da visão **Pedidos_v**, apresentado na Figura 5.1 e as ACs de **V** definidas na seção 5.1.5. Suponha que desejamos gerar o tradutor para inserções na visão **Pedidos_v**. Figura 5.25 mostra o tradutor para adições na visão **Pedidos_v**, gerado de acordo com o molde da Figura 5.24.

De acordo com o tradutor gerado, um pedido de adição de um objeto (:new) do tipo $\mathbf{T}_{pedido}$ na visão Pedidos_v é traduzido em uma atualização no banco de dados em dois passos.

No Passo 1, ocorre uma inserção da tupla $\mathbf{t}_{novo}$ = (new•codigo, $cliente_ref•codigo, :new•data, :new•dataEntrega, :new•enderecoEntrega•rua, :new•enderecoEntrega•cidade, :new•enderecoEntrega•estado, :new•enderecoEntrega•cep) na tabela **Pedido_rel**. Sempre que for realizada uma adição na visão **Pedidos_v**, o tradutor é disparado.

No Passo 2 são tratados os atributos multivalorados, cuja AC é da forma $\mathbf{T_v \bullet a}] \equiv [T_{R_b} \bullet \varphi]$, onde **a** é uma coleção aninhada de $\mathbf{T}_v$. Este é o caso do atributo multivalorado

listaitens da visão **Pedidos_v .** Neste passo, para cada objeto do tipo T_{item} na coleção **listaitens** é efetuada uma inserção na coleção aninhada **listaitens** do objeto **:new,** o qual foi inserido no passo 1**.** Dado que **codigo** é um identificador de **Pedidos_v** então temos que **v•codigo** = :new•**codigo**. Note que o pedido de atualização na coleção aninhada é traduzido em atualizações no banco de dados usando o tradutor para adição na coleção aninhada **listaitens** da visão **Pedidos_v**, mostrado na Figura 5.11.

A seguir mostramos como foram geradas, pelo procedimento **GVA**$_3$, as subconsultas que computam os valores dos atributos **pcodigo**, **pcliente**, **pdata**, **pdataentrega**, **prua**, **pcidade**, **pestado** e **pcep**.

- O atributo **pcodigo** é um atributo monovalorado de valor atômico. Do Caso 1 do procedimento **GVA**$_3$ e da ACC ψ_3: [T_{pedido_v}•**codigo**]≡ [$T_{pedidos\_rel}$•**pcodigo**], temos que $Q_{pcodigo}$ = :**new.codigo** ;
- O atributo **pdata** é um atributo monovalorado de valor atômico. Do Caso 1 do procedimento **GVA**$_3$ e da ACC ψ_4: [T_{pedido_v}•**data**] ≡ [$T_{pedidos\_rel}$•**pdata**], temos que Q_{pdata} = :**new.data** ;
- O atributo **pdataentrega** é um atributo monovalorado de valor atômico. Do Caso 1 do procedimento **GVA**$_3$ e da ACC ψ_5: [T_{pedido_v}•**dataEntrega**] ≡ [$T_{pedidos\_rel}$•**pdataEntrega**], temos que $Q_{pdataentrega}$ = :**new.dataentrega** ;
-O atributo **pcliente** é um atributo da chave estrangeira Fk1: Pedidos_rel[**pcliente**]⊂ Clientes_rel[**ccodigo**] e **cliente_ref** é um atributo de referência do tipo $T_{cliente}$. Nesse caso é realizada a chamada da função UTL_REF_SELECT OBJETC(**$cliente_ref**, :new.**cliente_ref**) no tradutor gerado. UTL_REF_SELECT OBJETCT é uma função do Oracle utilizada para "desreferenciar" um objeto, então temos que a variável **$cliente_ref** do tipo Tcliente recebe o objeto da referência ":new.**cliente_ref**". Dado que o caminho φ (**fk**$_1$) possui somente uma ligação, do Caso 5.2.1 procedimento **GVA**$_3$ e da ACC ψ7: [T_{pedido_v}•**cliente_ref**] ≡ [$T_{pedidos\_rel}$•**fk**$_1$] , temos que $Q_{pcliente}$= **$cliente_ref.codigo**;
- Os atributos **prua**, **pcidade**, **pestado** e **pcep** são monovalorados de valor atômico. Do caso 3 do procedimento **GVA**$_3$ e da ACC ψ_6: [T_{pedido_v}•**enderecoEntrega**], {**rua**, **cidade**, **estado**, **cep**}] ≡ [$T_{pedidos\_rel}$,{**prua**, **pcidade**], **pestado**, **pcep**}], temos que:

Q_{prua}=:new•enderecoEntrega•rua,

$Q_{pcidade}$=:new•enderecoEntrega•cidade,

$Q_{pestado}$=:new•enderecoEntrega•estado,

Q_{pcep} = :new• enderecoEntrega• cep.

```
CREATE OR REPLACE TRIGGER Adiciona_ Pedidos
INSTEAD OF INSERT ON Pedidos_v
DECLARE Lista_nt ListaItem; $cliente_ref  Tcliente;
BEGIN
Lista_nt := :new•listaItens;
UTL_REF_SelectObject($cliente_ref,:new•cliente_ref);

/*PASSO 1*/
Insert into Pedidos_rel
(pcodigo, pcliente, pdata, pdataentrega, prua, pcidade,pestado,pcep )
Values   (new•codigo,                                    (de ψ3)
          $cliente_ref•codigo,                           (de ψ7)
          :new•data,                                     (de ψ4)
          :new•dataEntrega,                              (de ψ5)
          :new•enderecoEntrega•rua,                      (de ψ6)
          :new•enderecoEntrega•cidade,                   (de ψ6)
          :new•enderecoEntrega•estado,                   (de ψ6)
          :new•enderecoEntrega•cep);                     (de ψ6)
/*PASSO 2*/
 FOR i in 1..Lista_nt.count LOOP
    Insert into Table( Select p•listaItens From Pedidos_v p   Where p•codigo = :new•codigo)
    Values(Lista_nt(i));
```

Figura 5.25: Tradutor Adiciona_ Pedidos

5.6 Definição de Tradutores para Operações de Remoção em Visões

Nesta seção apresentamos o algoritmo para geração dos tradutores para a operação de remoção em visões de objetos. Para remover um pedido da visão **Pedidos_v** da Figura 5.5, cujo código é 1, podemos usar a atualização mostrada na Figura 5.26. No Oracle 9i, **:old** refere-se ao objeto removido da visão **Pedidos_v**, isto é, o pedido de código=1.

```
DELETE FROM Pedidos_v v
WHERE v.código='1';
```

Figura 5.26: Exemplo de remoção na visão **Pedidos_v**

No resto desta seção, considere a visão **V** cujos objetos são do tipo T_v, como mostrado na Figura 5.23. Suponha que (i) a extensão da visão **V** é definida por uma ACE de

equivalência $\mathbf{V \equiv R_b}$ ou de subconjunto $\mathbf{V \subset R_b}$, onde $\mathbf{R_b}$ é a tabela base cujos objetos são do tipo $\mathbf{T_{Rb}}$ e (ii) $[\mathbf{T_v}, \{\mathbf{c_1}, \mathbf{c_2}, ..., \mathbf{c_w}\}] \equiv [\mathbf{T_{Rb}}\{\mathbf{d_1}, \mathbf{d_2}, ..., \mathbf{d_w}\}]$ é a ACO de $\mathbf{T_v}$ & $\mathbf{T_{Rb}}$.

Figura 5.27 mostra o molde do tradutor "RemoçãoNaVisaoV" para remoção na visão **V**. De acordo com o tradutor, um pedido de remoção de um objeto (:old) do tipo $\mathbf{T_v}$ na visão **V** é traduzido em uma remoção da tupla $\mathbf{t_{Rb}}$ na tabela R_b, tal que :old ≡ $\mathbf{t_{Rb}}$. Note que, de acordo com a ACO $[\mathbf{T_v}, \{\mathbf{c_1}, \mathbf{c_2}, ..., \mathbf{c_w}\} \equiv \mathbf{T_{Rb}}\{\mathbf{d_1}, \mathbf{d_2}, ..., \mathbf{d_w}\}]$ de $\mathbf{T_v}$ &$\mathbf{T_{Rb}}$, :old ≡ $\mathbf{t_{Rb}}$ sss "$\mathbf{t_{Rb}} \bullet \mathbf{d_k}$ =:old$\bullet \mathbf{c_k}$, $1 \leq k \leq w$. Na Seção A.4 do Anexo A mostramos que o tradutor gerado realiza a atualização solicitada na visão. A seguir mostramos um exemplo deste tradutor.

```
CREATE OR REPLACE TRIGGER RemoçãoNaVisaoV
INSTEAD OF DELETE ON V
BEGIN

DELETE FROM Rb tRb
WHERE  tRb•d1 = :old•c1 ...AND  tRb •dw = :old•cw

/*onde [Tv ,{c1,c2,...,cw} ≡ TRb {d1,d2,...,dw}] é a ACO de Tv & TRb */
END;
```

Figura 5.27: Tradutor para "RemoçãoNaVisãoV"

EXEMPLO 5.8:

Considere o esquema do banco de dados apresentado na Figura 5.2, o esquema da visão **Pedidos_v**, apresentado na Figura 5.1 e as ACs de **V** definidas na Seção 5.1.5. Suponha que desejamos gerar o tradutor para remoções na visão **Pedidos_v**. Figura 5.28 mostra o tradutor para remoções na visão **Pedidos_v** , gerado pelo algoritmo.

De acordo com o tradutor gerado, um pedido de remoção de um objeto (:old) do tipo $\mathbf{T_{pedido}}$ na visão **Pedidos_v** é traduzido em uma remoção da tupla $\mathbf{t_{pedidos\_rel}}$ na tabela **Pedidos_rel** , tal que :old ≡ $\mathbf{t_{pedidos\_rel}}$. Note que, de acordo com a ACO ψ_2: $[\mathbf{T_{pedido\_v}}, \{codigo\}] \equiv [\mathbf{T_{pedidos\_rel}}, \{pcodigo\}]$, :old ≡ $\mathbf{t_{pedidos\_rel}}$ sss "$\mathbf{t_{pedidos\_rel}}$.pcodigo =:old.codigo". Sempre que for realizada uma remoção na visão **Pedidos_v**, o tradutor é disparado.

```
CREATE OR REPLACE TRIGGER Remove_ Pedidos
INSTEAD OF DELETE ON Pedidos_v
BEGIN
DELETE FROM Pedidos_rel p
WHERE  p.pcodigo =:old.codigo;
END;
```

Figura 5.28: Tradutor Remove_ Pedidos

5.7 Definição de Tradutores para Operações de Modificação de Atributos Monovalorados da Visão

Nesta seção apresentamos o algoritmo para geração dos tradutores para a operação de modificação de atributo monovalorado da visão de objetos. Para modificar o valor do atributo **dataEntrega** da visão **Pedidos_v** da Figura 5.5, onde o código do pedido é 1, podemos usar a atualização mostrada na Figura 5.28. No Oracle 9i, **:new** refere-se ao objeto modificado na visão de objetos, que no exemplo é o pedido de código=1 da visão **Pedidos_v**.

```
UPDATE Pedidos_v v
SET v.dataEntrega='23/09/2004'
WHERE v.código='1';
```

Figura 5.28: Exemplo de Atualização na visão Pedidos_v

No resto desta seção, considere a visão **V** cujos objetos são do tipo T_v, como mostrado na Figura 5.23. Suponha que: (i) a extensão da visão **V** é definida por uma ACE de equivalência $V \equiv R_b$ ou de subconjunto $V \subset R_b$, onde R_b é a tabela base cujos objetos são do tipo T_{Rb} e (ii) $[T_v, \{c_1, c_2, ..., c_w\} \equiv T_{Rb} \{d_1, d_2, ..., d_w\}]$ é a ACO de T_v & T_{Rb}.

O algoritmo gera tradutores para os três casos descritos a seguir:

Caso 1: ACC é do tipo $[T_v \bullet a] \equiv [T_{Rb} \bullet b]$

Este caso trata a modificação de atributos monovalorados de valor, cuja ACC é do tipo $[T_v \bullet a] \equiv [T_{Rb} \bullet b]$, onde **a** é um atributo monovalorado e **b** é um atributo de T_{Rb}.

Figura 5.29 mostra o molde do tradutor "ModificaVisaoVCaso1" para este caso. De acordo com o tradutor, um pedido de modificação do atributo monovalorado **a** de um objeto (:new) do tipo T_v na visão **V** é traduzido em uma modificação do atributo **b** da tupla t_b na tabela R_b, tal que :new $\equiv t_b$. Note que, de acordo com a ACO $[T_v, \{c_1, c_2, ..., c_w\}] \equiv [T_{Rb} \{d_1, d_2, ..., d_w\}]$ de T_v & T_{Rb}, :new $\equiv t_b$ sss "$r_b \bullet d_k$ = :new$\bullet c_k$, $1 \leq k \leq w$. Na seção A.5.1 do Anexo A mostramos que o tradutor gerado realiza a atualização solicitada na visão.

```
CREATE OR REPLACE TRIGGER ModificaVisaoV
INSTEAD OF UPDATE ON V
BEGIN

UPDATE  Rb rb
SET rb•b = :new•a
WHERE  rb•d1 = :new•c1 ...AND rb•dw = :new•cw;
END;
```

Figura 5.29: Tradutor "ModificaVisãoVCaso1"

Caso2: ACC é do tipo $[T_v \bullet a,\{a_1,a_2,...,a_n\}] \equiv [T_{Rb},\{b_1,b_2, ..., b_n\}]$

Este caso trata a modificação de atributos monovalorados de valor, cuja ACC é do tipo $[T_v \bullet a,\{a_1,a_2,...,a_n\}] \equiv [T_{Rb},\{b_1,b_2, ..., b_n\}]$, onde **a** um atributo de T_v, cujo tipo T_a contém atributos atômicos $a_1,a_2,...,a_n$ e $b_1,b_2, ..., b_n$ atributos de valor atômico de T_{Rb}.

Figura 5.30 mostra o molde do tradutor "ModificaVisaoVCaso2" para este caso. De acordo com o tradutor, um pedido de modificação do atributo monovalorado estruturado **a,** onde **a** é um atributo de T_v, cujo tipo T_a contém atributos atômicos $a_1,a_2,...,a_n$ de um objeto (:new) do tipo T_v na visão **V,** é traduzido em modificações dos atributos de valor atômico $b_1,b_2, ..., b_n$ da tupla t_b na tabela R_b, tal que :new $\equiv t_b$. Note que, de acordo com a ACO $[T_v ,\{c_1,c_2,...,c_w\}] \equiv [T_{Rb} \{d_1,d_2,...,d_w\}]$ de T_v & T_{Rb}, :new $\equiv t_b$ sss "$r_b \bullet d_k$ = :new$\bullet c_k$, $1 \leq k \leq w$. Na Seção A.5.2 do Anexo A mostramos que o tradutor gerado realiza a atualização solicitada na visão. A seguir mostramos um exemplo deste tradutor.

```
CREATE OR REPLACE TRIGGER ModificaVisaoV
INSTEAD OF UPDATE ON V
BEGIN

UPDATE  Rb rb
SET rb•b1 = :new•a•a1 AND… rb•bn = :new•a•an
WHERE  rb•d1 = :new•c1 ...AND rb•dw = :new•cw;
END;
```

Figura 5.30: Tradutor "ModificaVisãoVCaso2"

EXEMPLO 5.9:

Considere o esquema relacional apresentado na Figura 5.2, o esquema da visão Pedidos_v, apresentado na Figura 5.1 e as ACs de Pedidos_v definidas na Seção 5.1.5.

Suponha que desejamos gerar o tradutor para modificação do atributo monovalorado **endereçoEntrega** da visão **Pedidos_v**. A Figura 5.31 mostra o tradutor para modificações de **endereçoEntrega** de um pedido em **Pedidos_v**.

De acordo com o tradutor gerado, um pedido de modificação do atributo monovalorado **endereçoEntrega**, cujo ACC é [$\mathbf{T}_{pedido\_v}$•**enderecoEntrega**], {**rua**, **cidade**, **estado**, **cep**}] ≡ [$\mathbf{T}_{pedidos\_rel}$,{**prua**, **pcidade**], **pestado**, **pcep**}], de um pedido em **Pedidos_v,** é traduzido em modificações dos atributos crua,ccidade,cestado,ccep da tupla $\mathbf{t}_{pedido\_rel}$ na tabela **Pedidos_rel**, tal que :new ≡ $\mathbf{t}_{pedido\_rel}$. Note que, de acordo com a ACO de $\mathbf{T}_{pedido\_v}$ & $\mathbf{T}_{pedido\_rel}$, "$\mathbf{t}_{pedido\_rel}$•**pcodigo** = :new•**codigo**". Sempre que for realizada uma modificação no atributo monovalorado **endereçoEntrega** da visão **Pedidos_v**, o tradutor é disparado.

```
CREATE OR REPLACE TRIGGER ModificaEnderecoEntregaVisaoPedidos_v
INSTEAD OF UPDATE ON V
BEGIN

UPDATE  Pedidos_rel t_pedido_rel
SET t_pedido_rel•crua = :new•endereçoEntrega•rua AND
      t_pedido_rel•ccidade = :new•endereçoEntrega•cidade AND
      t_pedido_rel•cestado = :new•endereçoEntrega•estado AND
      t_pedido_rel•ccep = :new•endereçoEntrega•cep
WHERE  t_pedido_rel•pcodigo = :new•codigo;
END;
```

Figura 5.31: Tradutor "ModificaEnderecoEntregaVisaoPedidos_v"

Caso 3: ACC é do tipo $\mathbf{T_v•a ≡ T_{Rb}•φ• k}$

Este caso trata a modificação de atributos monovalorados de valor, cuja ACC é do tipo **[$\mathbf{T_v•a}$] ≡[$\mathbf{T_{Rb}•φ}$]**, onde **a** é um atributo monovalorado e φ é um caminho de $\mathbf{T_{Rb}}$.

Neste caso, como mostrado em [37], para garantirmos que não existe ambigüidade no nível de dados, as seguintes condições devem ser satisfeitas:

1. A assertiva de correspondência do atributo monovalorado **a** é dado por: $\mathbf{T_v.a ≡ T_{Rb}.φ}$, onde φ é um caminho de $\mathbf{T_{Rb}}$ definido por $φ=\ell_1•\ell_2•..•\ell_j•..•\ell_n$ (Figura 5.32), tal que:
   1.1. ℓ_1 é uma ligação de chave estrangeira dada por $\mathbf{R_b}$ [$\mathbf{f}_1^{\ell_1}$, ..., $\mathbf{f}_{m_1}^{\ell_1}$] ⊂ $\mathbf{R}_{\ell_1}$ [$\mathbf{g}_1^{\ell_1}$,.., $\mathbf{g}_{m_1}^{\ell_1}$] ou inversa da chave estrangeira dada por $\mathbf{R}_{\ell_1}$ [$\mathbf{g}_1^{\ell_1}$, ...,$\mathbf{g}_{m_1}^{\ell_1}$] ⊂ $\mathbf{R_b}$ [$\mathbf{f}_1^{\ell_1}$, ..., $\mathbf{f}_{m_1}^{\ell_1}$]; e ℓ_i , $2 \leq i \leq n$, é uma ligação de chave estrangeira dada por $\mathbf{R}_{\ell_{i-1}}$ [$\mathbf{f}_1^{\ell_i}$, ...,

$\mathbf{f}_{m_i}{}^{\ell_i}] \subset \mathbf{R}_{\ell_i}[\mathbf{g}_1{}^{\ell_i},.., \mathbf{g}_{m_i}{}^{\ell_i}]$ ou inversa da chave estrangeira dada por $\mathbf{R}_{\ell_i}[\mathbf{g}_1{}^{\ell_i}, ...,\mathbf{g}_{m_i}{}^{\ell_i}] \subset \mathbf{R}_{\ell_{i-1}}[\mathbf{f}_1{}^{\ell_i}, ...,\mathbf{f}_{m_i}{}^{\ell_i}]$.

1.2. Considere ℓ_{j+1}, $1 \leq i \leq n$, a ligação monovalorada direta onde será realizada a atualização, a qual é definida em tempo de projeto. Só poderá existir no máximo uma ligação ℓ_i no caminho φ cuja inversa é multivalorada. Caso contrário, como mostrado em [37], existe ambigüidade no nível de dados e o tradutor não poderá ser definido em tempo de projeto. No caso em que φ possui uma ligação cuja inversa é multivalorada, então esta ligação deve ser escolhida, uma vez que se uma das outras ligações de φ fosse escolhida existiria ambigüidade [37]. No caso de todas as inversas das ligações serem monovaloradas, então qualquer uma das ligações do caminho pode ser escolhida. Portanto o projetista deve ser consultado para determinar a ligação em tempo de projeto. Assim, temos que as ligações $\ell_1,.., \ell_j$ e $\ell_{j+2},.., \ell_n$ e suas inversas são monovaloradas e a inversa de ℓ_j pode ser multivalorada ou monovalorada. Na nossa abordagem, não permitimos a modificação de atributos que sejam identificadores, portanto ℓ_{j+1} não é um identificador de $\mathbf{R}_{\ell_j}$.

1.3. **k** é um atributo de $\mathbf{T}_{R_{\ell_n}}$, tal que **k** é um identificador de $\mathbf{R}_{\ell_n}$. Esta condição garante que com o valor de **k** iremos recuperar apenas uma instância de $\mathbf{R}_{\ell_n}$.

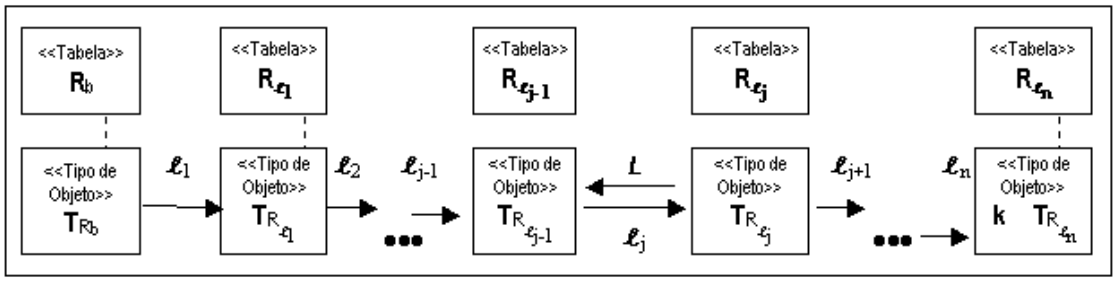


Figura 5.32: Caminho $\ell_1 \bullet \ell_2 \bullet .. \bullet \ell_{j-1} \bullet .. \bullet \ell_n \bullet \mathbf{k}$ do tipo base $\mathbf{T}_{Rb}$

A Figura 5.33 mostra o molde do tradutor "ModificaVisaoV" para este caso. De acordo com o tradutor, um pedido de modificação do atributo monovalorado **a** de um objeto (:new) do tipo $\mathbf{T}_v$ na visão **V** é traduzido em uma modificação dos atributos da chave estrangeira ℓ_{j+1} da tupla $\mathbf{t}_{R\ell_j}$,onde $\mathbf{t}_{R\ell_j} = \mathbf{t}_{Rb} \bullet \ell_1 \bullet .. \bullet \ell_j$ e $\mathbf{t}_{Rb} \equiv$:new. Note que, de acordo com a

ACO de $\mathbf{T}_v$ & $\mathbf{T}_{R_b}$ $[T_{R_b},\{\mathbf{c}_1,\mathbf{c}_2,...,\mathbf{c}_w\}]\equiv[T_v,\{\mathbf{d}_1,\mathbf{d}_2,...,\mathbf{d}_w\}]$, :new ≡ $\mathbf{t}_{Rb}$ sss "$\mathbf{t}_{Rb}\bullet\mathbf{c}_i$ = :new•$\mathbf{d}_i$, 1≤ i ≤ w. Na Seção A.5.3 do Anexo A, mostramos que do novo valor atribuído a ℓ_{j+1}, temos que $\mathbf{t}_{R\ell_j}\bullet\ell_{j+1} = \mathbf{t}_{R\ell_{j+1}}$, onde $\mathbf{t}_{R\ell_{j+1}}\bullet\ell_{j+1}\bullet..\bullet\ell_n\bullet\mathbf{k}$ = :new.**a**. Assim temos que $\mathbf{T_{Rb}}\bullet\varphi\bullet\mathbf{k}$ =**:new•a**. A seguir mostramos um exemplo deste tradutor.

```
CREATE OR REPLACE TRIGGER ModificaVisaoVCaso3
INSTEAD OF UPDATE ON V
BEGIN
UPDATE R_ℓj t_Rℓj
SET t_Rℓj•f_s^ℓj+1 = (Select t_Rℓj+1•g_s^ℓj+1
                      From R_ℓj+1 t_Rℓj+1 ,....., R_ℓn t_Rℓn
                      Where t_Rℓj+1•f_q^ℓj+2= t_Rℓj+2•g_q^ℓj+2 , 1≤ k≤ m_j+2 AND t_Rℓp-1•f_q^ℓp= t_Rℓp•g_q^ℓp ,
                            1≤ q≤ m_p , j+1≤p≤n  AND t_Rℓn•k= :new•a ) , 1≤ s ≤ m_j+1
WHERE t_Rℓj•g_s^ℓj = (Select t_Rℓj-1•f_s^ℓj
                       From  R_b t_Rb ,....., R_ℓj-1 t_Rℓj-1
                       Where t_Rb•f_q^ℓ1= t_Rℓ1•g_q^ℓ1, 1≤ q≤m_1 and t_Rℓk-1•f_q^ℓk=t_Rℓk•g_q^ℓk, 1≤q≤m_k ,
                       2≤k≤j-1 and t_Rb•c_1 = :new•d_1 ...AND t_Rb•c_w= :new• d_w), 1≤ s ≤ m_j
END;
```

Figura 5.33: Tradutor "ModificaVisãoVCaso3"

EXEMPLO 5.10:

Considere o esquema do banco de dados apresentado na Figura 5.34 (b), o esquema da visão **Gerentes_v**, apresentado na Figura 5.34 (a) e as ACs de **V** definidas abaixo.

Suponha que desejamos gerar o tradutor para modificação do atributo monovalorado **#proj_v** da visão **Pedidos_v**. Da AC de **#proj_v** [**Tgerente_v•#proj_v**] ≡ [**Tgerente•**$FK_1\bullet FK_2$**•#proj_v**], temos que neste exemplo, como todas as inversas das ligações do caminho $FK_1\bullet FK_2$ são monovaloradas então qualquer uma das ligações pode ser escolhida para ser realizada a atualização. Suponha que a ligação FK_1 do caminho $FK_1\bullet FK_2$ foi escolhida. Figura 5.35 mostra o tradutor para modificação do atributo **#proj_v** de Gerentes_v, gerado pelo molde da Figura 5.33.

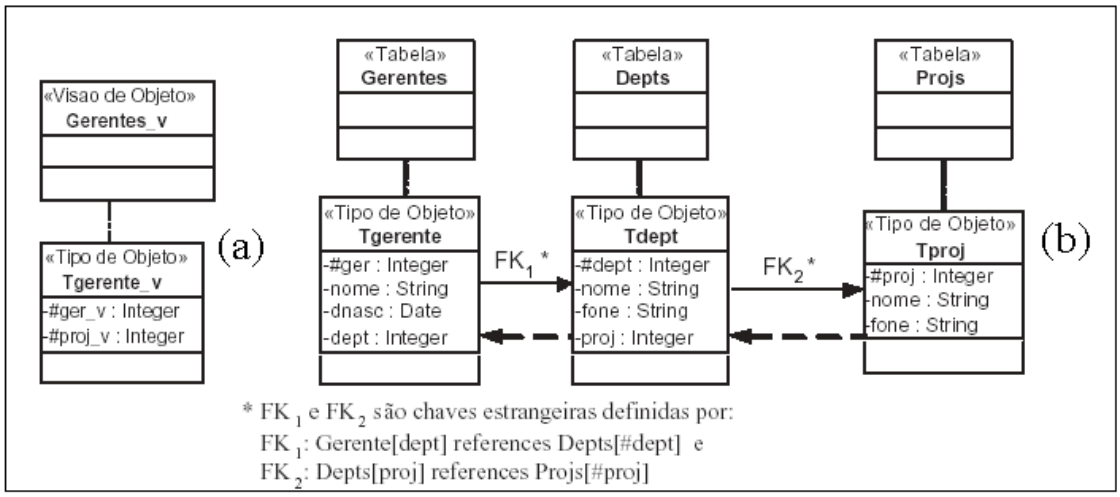


Figura 5.34: Esquema do Banco de Dados (a) da Visão de Objetos **Gerentes_v** (b)

Assertiva de Correspondência de Extensão:

ψ_1: [**Gerentes_v**]≡ [**Gerentes**]

Assertivas de Correspondência de Objetos:

ψ_2: [**Tgerente_v**, {**#ger_v**}] ≡ [**Tgerente**, {**#ger**}]

Assertivas de Correspondência de Caminho

ψ_3: [**Tgerente_v•#proj_v**] ≡ [**Tgerente**•FK_1•FK_2•**#proj_v**]

ψ_4: [**Tgerente-v•#ger-v**] ≡ [**Tgerente•#ger**]

```
CREATE OR REPLACE TRIGGER ModificaProjeto
INSTEAD OF UPDATE ON Gerentes_v
BEGIN

UPDATE  Gerentes g
SET  g.dept = ( Select d.#dept
                From Depts, Projs p
                Where p.#proj=:new.pro_v and d.proj = p.#proj)

WHERE g.#ger = :new.#ger_v ;
END;
```

Figura 5.35: Tradutor "ModificaProjeto"

De acordo com o tradutor gerado, um pedido de modificação do atributo monovalorado **#proj_v** de um objeto (:new) na visão **Gerentes_v** é traduzido na modificação do valor do atributo #dept (de FK_1) da tupla **g** em Gerentes, tal que **g** ≡:new (da ACO

[**Tgerente_v**, {#**ger_v**}] ≡ [**Tgerente**, {#**ger**}]). Do novo valor atribuído, temos que **g**•FK_1•FK_2•**#proj_v** = :new.**#proj_v**. Sempre que for realizada uma modificação no atributo monovalorado **#proj-v** da visão Gerentes_v, o tradutor é disparado.

Capítulo 6 – *XUpdateTranslator* - O Tradutor de Atualizações do XML Publisher

Neste capítulo, discutimos a implementação do *XUpdateTranslator*, o tradutor de atualizações *XQuery* do XML *Publisher*. Inicialmente, descrevemos as fases de tradução de atualizações no XML *Publisher*. Na Seção 6.2, abordamos os tipos de atualização XQuery que podem ser traduzidas no *XUpdateTranslator*. Os tipos de atualizações que podem ser traduzidas foram definidos na extensão da linguagem de consulta XQuery proposta em [16]. Nas demais seções, apresentamos os algoritmos que implementam as funcões do *XUpdateTranslator*.

6.1 Tradução de atualizações no XUpdateTranslator

Como discutido anteriormente, as visões XML publicadas no XML *Publisher* podem ser atualizadas usando a extensão da linguagem *XQuery* proposta em [16]. A tradução de atualizações *XQuery* no XML *Publisher* é realizada pelo *XUpdateTranslator*. Basicamente, o *XUpdateTranslator* é um software para tradução de atualizações *XQuery* em SQL3. Como ilustrado na Figura 6.1, a arquitetura do *XUpdateTranslator* é composta pelos módulos Validador e Tradutor, sendo que as fases envolvidos na tradução de atualizações *XQuery* no *XUpdateTranslator* são: Parsing, Tradução, e Execução.

6.1.1 Fase de *Parsing*

No *XUpdateTranslator*, assim como no processamento de atualizações em SGBD's convencionais, na fase de *parsing* ocorre a análise léxica e sintática da atualização *XQuery*. Em seguida, ocorre a análise semântica, ou validação, onde são verificados se todas as expressões de caminhos (*XPath*) da atualização são válidas e semanticamente significativas para o XML *Schema* da respectiva visão XML. A validação nos permite avaliar com mais precisão erros de sintaxe e semântica na atualização *XQuery* de entrada. Se a validação não existisse, a atualização *XQuery* seria traduzida erroneamente em SQL3 durante a fase de

tradução. Sendo assim, os erros seriam detectados durante a fase de execução da atualização SQL3 e a mensagem de erro gerada pelo SGBD talvez não fosse suficiente para depurar o erro na *XQuery*.

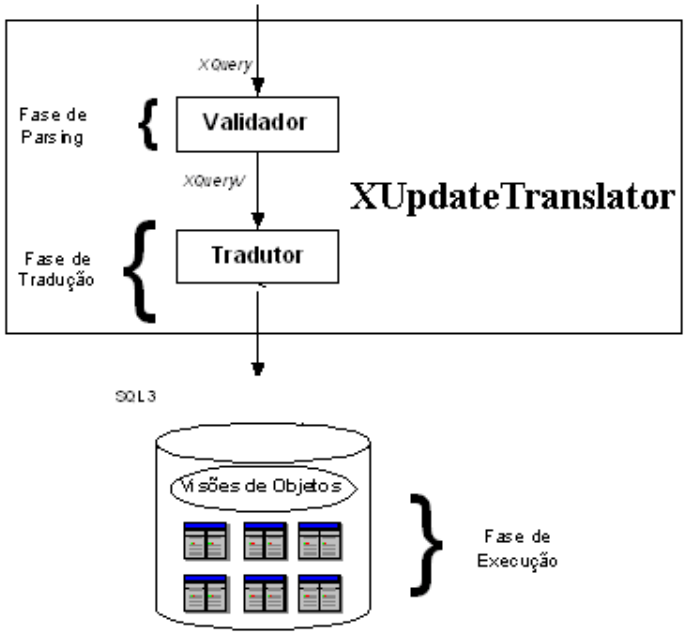


Figura 6.1: Arquitetura do ***XUpdateTranslator.***

A fase de *parsing* no *XUpdateTranslator* acontece no módulo Validador. Sua implementação não é discutida nesta dissertação. Maiores detalhes sobre as análises léxica, sintática e semântica podem ser encontrados em textos especializados em compiladores [59]. Para este trabalho, é suficiente saber que toda atualização *XQuery* validada corretamente pelo módulo Validador é denominada atualização *XQuery* Válida (XQueryV). Na Seção 6.2 descrevemos em que condições uma atualização *XQuery* é válida.

6.1.2 Fase de Tradução

Na fase de tradução a atualização é traduzida numa atualização SQL3, que é a linguagem de consulta e atualização para bancos de dados objeto-relacionais. A fase de tradução no *XUpdateTranslator* é executada no módulo Tradutor, no qual a atualização *XQueryV* é diretamente traduzida em uma seqüência de atualizações SQL3 sobre a VOD. Nessa tradução, as cláusulas *for* e *where* do comando de atualização *XQueryV* são diretamente mapeadas nas cláusulas *FROM, WHERE*, respectivamente, do comando de

atualização SQL3. As cláusulas *update insert*, *update delete* e *update replace* são mapeadas em cláusulas de atualização SQL3. Na Seção 6.3 discutimos detalhadamente os algoritmos de tradução.

6.1.3 Fase de Execução

A fase de execução da atualização *XQuery* no *XUpdateTranslator* corresponde então ao processamento das atualizações SQL3 nas Visões de Objetos do SGBD. As atualizações na Visões de Objetos são por sua vez traduzidas pelos *"instead od triggers"*, em atualizações definidas sobre o esquema do banco de dados, como definido no Capítulo 5. Os *"instead of triggers"* das Visões de Objetos são gerados em tempo de projeto.

6.2 Atualizações XQuery Válidas

A linguagem de consulta aceita pelo XML *Publisher* é a XQuery. Essa linguagem foi escolhida por ser largamente aceita, além do que está se tornando um padrão para consulta a dados XML. Neste trabalho, utilizamos a extensão da linguagem de consulta XQuery proposta em [18]. Esta extensão corresponde à implementação dos operadores que são necessários para efetuarmos operações de atualização em visões XML. São definidos os seguintes operadores: insert (inserção), delete (remoção) e replace (modificação).

XQuery foi estendida com a seguinte estrutura para atualizações: FOR.... WHERE (opcional)...UPDATEOP (Figura 6.2 (a)). Sendo especificada uma seqüência de sub-operações na cláusula UPDATEOP (Figura 6.2 (b)). As atualizações que podem ser processadas no *XUpdateTranslator* são chamadas de atualizações *XQuery* Válidas (XQueryV), ou seja, atualizações que podem ser validadas pelo módulo Validador.

FOR v_1$ IN $V/$\delta v$_1$, v_2$in $v1/$\delta v$_2$, ….,v_n$ in v_{n-1}$/$\delta v$_n$ WHERE Condv$_1$, Condv$_2$....., Condv$_{n-1}$ UPDATEOP	UPDATE v_n$ {**subOp,** {**subOp**}*} E **subOp** é: DELETE $filho | INSERT $filho | REPLACE $filho WITH $conteudo
(a)	**(b)**

Figura 6.2- Operadores da extensão XQuery

Neste trabalho, utilizamos comandos de atualização XQuery que possuam uma expressão FWU (FOR WHERE UPDATE) contém uma ou mais variáveis, cada variável v_i, $1 \leq i \leq n$, tem associado:

- Uma expressão de caminho $\delta v_i = e_1/e_2/...../e_p$, $p>0$, onde $e_1, e_2, ..., e_{p-1}$ são elementos monocorrência e $\mathbf{e_p}$ pode ser monocorrência(v_n) ou multiocorrência ($v_1,...,v_n$);
- Uma condição $Condv_i$, a qual é opcional.

Exemplo 6.1: Inserção XQuery

No exemplo a seguir considere a visão de XML apresentada na Figura 6.6. A atualização *XQueryV* é uma inserção (INSERT) de um item para o Pedido cujo cliente é "Wamberg". $V é a visão Pedidos_v, os caminhos $\delta v_{1=}$ "Pedidos/Pedido" e δv_2 = "listaItens" são associados, respectivamente, as variáveis $\$v_1$ e $\$v_2$, $Condv_1$ é a condição associada à variável $\$v_1$ e $filho é o elemento item.

```
for $v1 in view("Pedidos_v")/Pedidos/Pedido,
    $v2 in $v1/listaItens
where $v1/cliente = "Wamberg"
update $v2
insert
  <item>
    <codigo>10</codigo>
    <produto>Impressora</produto>      } $filho
    <quantidade>100</quantidade>
  </item>
```

Figura 6.3 – Atualização XQuery **Q_1**

6.3 Módulo Tradutor

A fase de tradução no *XUpdateTranslator* ocorre no Módulo Tradutor. Sua função é traduzir cada atualização XQueryV em atualizações SQL3 sobre a respectiva visão de objetos *default*. Essa tradução é realizada de uma forma direta, na qual as cláusulas *for* e *where* da atualização XQueryV são diretamente mapeadas nas cláusulas *FROM, WHERE*, respectivamente, da atualização SQL3 correspondente. As cláusulas *update insert*, *update delete* e *update replace* são mapeadas em cláusulas de atualização SQL3 correspondente.

A seguir apresentamos os algoritmos para realizar essa tradução. Os algoritmos que implementam as funcões do *XUpdateTranslator* recebem como parâmetro de entrada uma operação de atualização XQuery e traduzem esta atualização em uma seqüência de atualizações SQL3 a serem executadas nas Visões de Objetos. O restante do capítulo está dividido como se segue. Na Seção 6.3.1 discutimos o algoritmo da operação de inserção

XQuery. Na Seção 6.3.2 discutimos o algoritmo da operação de remoção XQuery. Finalmente, na Seção 6.3.3, discutimos o algoritmo da operação de modificação XQuery.

6.3.1 Algoritmo para Traduzir uma Inserção *XQueryV* em Inserções ou Modificações SQL3

Nesta seção apresentamos o algoritmo implementado no módulo Tradutor do *XUpdateTranslator* para transformar inserções *XQueryV* de uma visão XML V_x, em atualizações SQL3 sobre a respectiva visão de objetos *default* VO, a partir do esquema de V_x.

Como apresentado na Figura 6.4, o algoritmo **TraduzInserçãoXQuery** recebe como entrada uma atualização *XQueryV* e o XML *Schema* da respectiva visão XML. O algoritmo **TraduzInserçãoXQuery** transforma inserções *XQueryV* de uma visão XML V_x, em atualizações SQL3 sobre a respectiva visão de objetos *default* VO para os três casos apresentados na Figura 6.4. A seguir, descrevemos cada um destes casos.

Algoritmo TraduzInserçãoXQuery (**XQ**: Expressão *XQueryV*)
φ =∅;
Seja **δ** = $\delta v_1/\delta v_2/\ldots./\delta v_n$ o caminho da raiz até δv_n e Seja ***S*** o esquema default de objetos
Caso 1:O caminho δ/label ($filho) corresponde a uma visão de objeto **VO** em ***S***
Caso 2: O caminho δ/label ($filho) corresponde a uma coleção aninhada **NT** em ***S***
Caso 3: O caminho δ/label ($filho) corresponde a um atributo monovalorado em ***S***

Figura 6.4: Algoritmo **TraduzInserçãoXQuery**

Caso 1:

Neste caso, uma inserção *XQueryV* é traduzida em uma inserção em uma visão de objetos VO, isto é, o caminho δ/label($filho)[4] corresponde a uma visão de objeto VO em *S*. Essa correspondência é obtida na geração do esquema da VOD, o qual faz um mapeamento entre os elementos do esquema da visão XML e o esquema da VOD, como definido no Capítulo 4.

Como apresentado na Figura 6.5, o Caso 1 do algoritmo **TraduzInserçãoXQuery** é dividido em dois subcasos.

[4] label($filho) é uma função que recebe como parâmetro de entrada $filho, ou seja, o elemento a ser inserido, e retorna o nome deste elemento.

Caso 1:O caminho δ/label ($filho) corresponde a uma visão de objeto **VO** em ***S***
  Caso 1.1: N° de aninhamento de coleções aninhadas de valor em $T_{\$filho} \leq 1$
    φ= INSERT INTO **VO** VALUES (**CriaObjeto1** (**$T_{\$filho}$,$filho**));
  Caso 1.2: N° de aninhamento de coleções aninhadas de valor em $T_{\$filho} > 1$
    φ=INSERT INTO **VO**VALUES (**CriaObjeto2**(**$T_{\$filho}$,$filho**));+**CriaInserção**($T_{\$filho}$,**$filho**);

Figura 6.5: Caso 1 do Algoritmo **TraduzInserçãoXQuery**

Caso 1.1:

No Caso 1.1, o número de aninhamentos de coleções aninhadas de valor em $T_{\$filho}$ é menor ou igual a 1, onde $T_{\$filho}$ é o tipo do elemento a ser inserido ($filho).

Observe que no Caso 1.1 é efetuada uma chamada ao procedimento CriaObjeto1. Esse algoritmo recebe como entrada $T_{\$filho}$ e $filho, onde $T_{\$filho}$ define o tipo do objeto a ser criado e $filho seus valores. Os valores obtidos pelo procedimento são definidos efetuando uma chamada ao procedimento Valor (Tabela 6.2). O objeto criado pelo procedimento CriaObjeto1 é inserido na visão de objetos VO. As tabelas a seguir mostram como este objeto é criado. O procedimento CriaObjeto1 também é utilizado nos casos 2 e 3, como veremos a seguir.

Casos	**CriaObjeto1($T_{\$filho}$, $filho)**
1. **$T_{\$filho}$** é um ComplexType, label($filho) e parent[5](label($filho) são multiocorrências Seja **$T_{\$filho}$**_item o tipo dos objetos de **$T_{\$filho}$** no Esquema Default de Objetos (EDO) Seja $e_1, e_2,...,e_n$ os elementos de **$T_{\$filho}$** em **$filho**	**$T_{\$filho}$**_item (valor ($e_1$), valor ($e_2$),...,valor ($e_n$))
2. **$T_{\$filho}$** é um ComplexType Seja $e_1, e_2,...,e_n$ os elementos de **$T_{\$filho}$** em **$filho**	**$T_{\$filho}$** (valor (e1), valor (e2),...,valor (e_n))
3. **$T_{\$filho}$** é um Simple Type	**$filho.getvalue()**
4. **$T_{\$filho}$** é uma referência para **T'**, onde T' é um tipo do Esquema Default de Objetos (EDO) /* $V' é a visão de objetos associada a T' e OID é o identificador único (object identifier) da visão $V'	Select REF(v) From $V' v where **OID** = **$filho.getvalue()**

Tabela 6.1: Procedimento **CriaObjeto1**

Casos	**Valor(e_i)**
1. **e_i** é multiocorrência, possui irmãos e **e_i** é uma referência para um elemento do tipo T', onde T' é um tipo do Esquema Default de Objetos (EDO) Seja T_{e_i} o tipo de **e_i** Seja $S_1, $S_2,...,$s_n$ instâncias de T_{e_i}	**T'**(CriaObjeto1(T_{e_i}, $S_1), CriaObjeto1($T_{e_i}$, $S_2),..., CriaObjeto1($T_{e_i}$, $S_n))
2. **e_i** é multiocorrência, possui irmãos e T_{e_i} não é uma referência para T'	**T_{e_i}**(CriaObjeto1(T_{e_i}, $S_1), CriaObjeto1($T_{e_i}$, $S_2),..., CriaObjeto1($T_{e_i}$, $S_n))

[5] parent é uma função que recebe com parâmetro de entrada um elemento e retorna o seu elemento pai.

Seja T_{e_i} o tipo de $\mathbf{e_i}$ Seja $\$S_1$, $\$S_2$,...,$\$s_n$ instâncias de T_{e_i}	
3. $\mathbf{e_i}$ é multiocorrência e não possui irmãos Seja T_{e_i} o tipo de $\mathbf{e_i}$ Seja $\$S_1$, $\$S_2$,...,$\$s_n$ instâncias de T_{e_i}	CriaObjeto1(T_{e_i}, $\$S_1$), CriaObjeto1($T_{e_i}$, $\$S_2$),..., CriaObjeto1($T_{e_i}$, $\$S_n$)
4. $\mathbf{e_i}$ é monocorrência Seja T_{e_i} o tipo de $\mathbf{e_i}$ Seja $\$S_1$, $\$S_2$,...,$\$s_n$ instâncias de T_{e_i}	CriaObjeto1(T_{e_i}, $\$S_1$)

Tabela 6.2: Casos do procedimento **Valor**(e_i) para **CriaObjeto1**

Observe que nos Casos 3 e 4 (Tabela 6.1) o objeto criado é obtido de forma direta. No entanto, nos Casos 1 e 2 o esquema da visão XML precisa ser analisado para determinarmos os valores do objeto criado. Sendo assim, os Casos 1 e 2 fazem chamadas recursivas ao procedimento **CriaObjeto1** (Tabela 6.2).

Nos exemplos apresentados neste capítulo, considere o esquema da visão XML Pedidos_v representado na Figura 6.6 (a) e o esquema da sua respectiva VOD, mostrado na Figura 6.6 (b). Para essa discussão o esquema do banco de dados pode ser ignorado uma vez que a execução da atualização SQL3 na VOD é traduzida em atualizações nas tabelas do banco de dados pelos Triggers INSTEAD OF. A geração semi-automática dos tradutores de atualização foi discutida no Capítulo 5.

Exemplo 6.2:

Considere a atualização $\mathbf{A_1}$ (Figura 6.7 (a)) que insere um elemento Pedido na Visão XML Pedidos_v. A atualização *XQuery* $\mathbf{A_1}$ será traduzida na atualização SQL3 da Figura 6.7 (b). Dado que δv_1= Pedidos e label($filho)=Pedido, de acordo com os mapeamentos obtidos durante a geração do esquema da VOD, temos que o caminho δv_1/ label($filho) corresponde à visão de objetos **Pedidos_v**.

O número de aninhamentos de coleções aninhadas na estrutura do tipo Tpedido ($T_{\$filho}$ = Tpedido) (Figura 6.7 (a)) é igual a 1, pois, o elemento item é o único elemento multiocorrência que descende de Tpedido. E, seja Titem o tipo do elemento item, a estrutura do tipo Titem não possui elementos multiocorrência. Assim, temos uma inserção na visão de objetos **Pedidos_v**

O procedimento **CriaObjeto1** gera, a partir dos valores dos elementos do tipo Tpedido da visão XML (Linhas 4 a 27), os valores dos atributos do tipo Tpedido da visão de

objetos **Pedidos_v** (Linhas 3 a 16). Dado que Tpedido é um ComplexType e parent(Pedido)= Pedidos não é multiocorrência, pelo Caso 2 do procedimento **CriaObjeto1**, o tipo do objeto a ser inserido na visão de objetos é Tpedido. Os elementos código, data e dataEntrega são monocorrência e seus tipos são SimpleType, assim, pelo Caso 3 do procedimento **CriaObjeto1**, os valores são mapeados, respectivamente, para as linhas 4, 5 e 6 da Figura 6.7(b). O elemento enderecoEntrega é monocorrência e seu tipo é um ComplexType, assim os valores dos elementos monocorrência rua, cidade, cep e estado são mapeados para linha 7 da Figura 6.7(b).

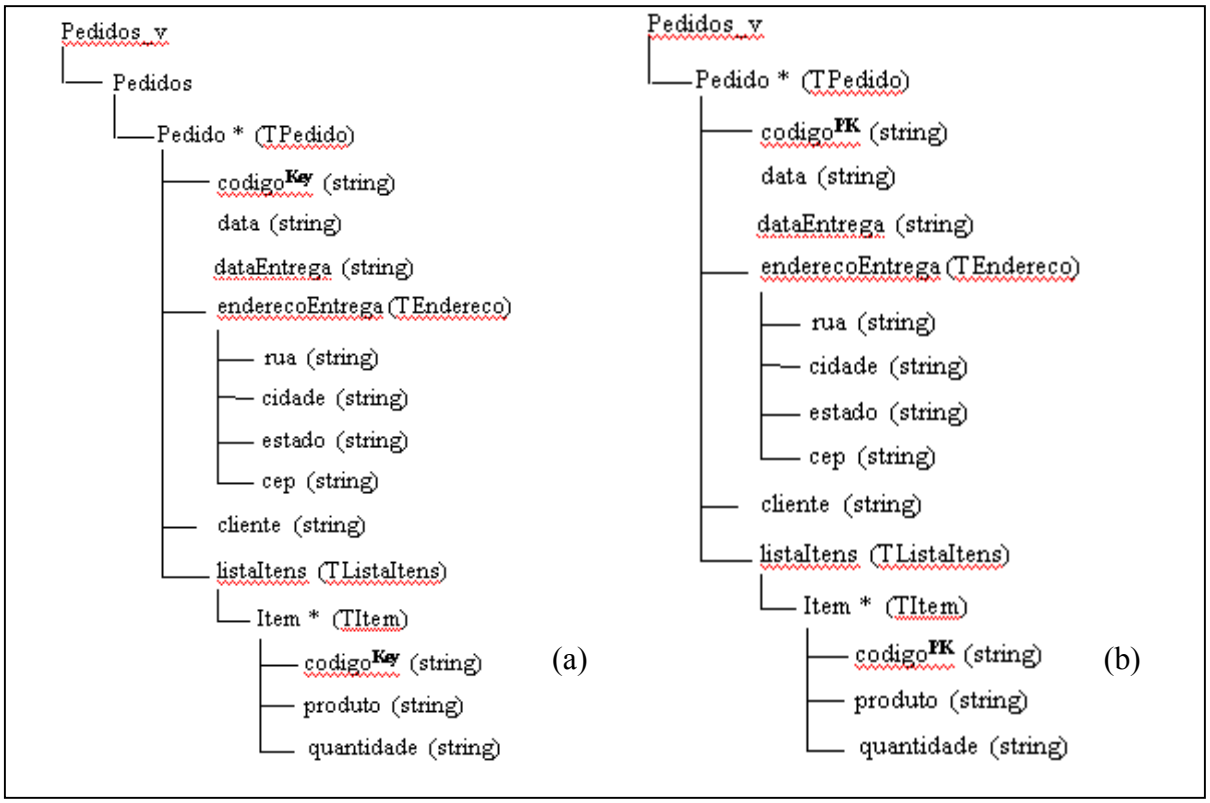


Figura 6.6: Esquemas da visão XML **Pedidos** (a) e da visão de objetos **Pedidos_v** (b)

O elemento listaitens é monocorrência e seu tipo TListaItens é um ComplexType (linha 9 da Figura 6.7(b)). Como TlistaItens possui somente o elemento item, que é multiocorrência, e o tipo do elemento item é Titem (ComplexType) (linhas 9 e 13 da Figura 6.7(b)), o Caso 2 do procedimento **CriaObjeto1** deve ser aplicado. Titem contém os elementos monocorrência codigo, produto e quantidade, cujos tipos são SimpleType. Os valores destes elementos são mapeados para linha 9 a 16 da Figura 6.7(b). Neste caso, os valores são obtidos através do Caso 3 do procedimento **CriaObjeto1**.

```
1.  For $v1 in view("Pedidos")/Pedidos
2.  update $v1
3.  insert
4.   <Pedido>
5.    <codigo>2</codigo>
6.    <data>10/11/2002</data>
7.    <dataEntrega>12/12/2002</dataEntrega>
8.    <enderecoEntrega>
9.      <rua>rua b</rua>
10.     <cidade>sobral</cidade>
11.     <estado>CE</estado>
12.     <cep>60000660</cep>
13.   </enderecoEntrega>
14.   <cliente>Fernando</cliente>
15.   <listaItens>
16.     <item>
17.       <codigo>10</codigo>
18.       <produto>Impressora</produto>
19.       <quantidade>100</quantidade>
20.     </item>
21.     <item>
22.       <codigo>14</codigo>
23.       <produto>Placa-som</produto>
24.       <quantidade>80</quantidade>
25.     </item>
26.   </listaItens>
27. </Pedido>
```

(A)

```
INSERT INTO Pedidos_v v1
(código,data, dataEntrega, enderecoEntrega, cliente,
listaItens)
VALUES( TPedido(
                '2',
                '10/11/2002',
                '12/12/2002',
                Tendereco('rua b', 'sobral','CE', '6000'),
                'Fernando',
                TlistaItens(
                   Titem('10',
                         'Impressora',
                         '100'),
                          )
                      )
                 );
```

(B)

Figura 6.7: Atualização *XQueryVC* $\mathbf{A_1}$ (A) e tradução SQL3 de $\mathbf{A_1}$ (B)

Caso 1.2:

No Caso 1.2, o número de aninhamentos de coleções aninhadas de valor em $T_{\$filho}$ é maior que 1, onde $T_{\$filho}$ é o tipo do elemento a ser inserido ($filho).

Observe que o no Caso 1.2 é efetuada uma chamada ao procedimento CriaObjeto2 (Tabela 6.3). Esse procedimento recebe como entrada $T_{\$filho}$ e $filho, onde $T_{\$filho}$ define o tipo do objeto a ser criado e $filho seus valores. O procedimento CriaObjeto2 cria parcialmente o objeto a ser inserido na visão de objetos VO, isto é, trata somente os valores para os elementos monocorrência. A Tabela 6.3 mostra como este objeto é criado

Casos **Seja $e_1,e_2,...,e_n$ os elementos de $T_{\$filho}$ em $filho**	**CriaObjeto2($T_{\$filho}$, $filho)**
1. $\mathbf{e_i}$ é monocorrência, possui irmãos e $\mathbf{e_i}$ é uma referência para um elemento do tipo T', onde T' é um tipo do Esquema Default de Objetos (EDO) Seja T_{e_i} o tipo de $\mathbf{e_i}$	Select REF(v) From $V' v where **OID** = **$e_i$.getvalue();** /* Onde $V' é a visão de objetos associada a T' e **OID** é o identificador único (object identifier) da visão $V' */

Seja $\$S_1$ a instância de T_{e_i}	
2. $\mathbf{e_i}$ é monocorrência e T_{e_i} é um SimpleType, onde T_{e_i} é o tipo de $\mathbf{e_i}$	$\mathbf{e_i}$**.getvalue()**
3. $\mathbf{e_i}$ é monocorrência e T_{e_i} é um ComplexType, onde T_{e_i} é o tipo de $\mathbf{e_i}$ Seja $\$S_1$ a instância de T_{e_i}	**CriaObjeto1**(T_{e_i}, $\$S_1$)
4. $\mathbf{e_i}$ é multiocorrência	T_{e_i}( )

Tabela 6.3: Procedimento **CriaObjeto2**

No Caso 1.2 (Figura 6.3) também é efetuada uma chamada ao procedimento **CriaInserção** (Tabelas 6.4 ,6.5 e 6.6). Esse procedimento recebe como entrada $\mathbf{T_{\$filho}}$ e **$filho**. Então, o procedimento gera comandos de inserção nas visões de objetos e coleções aninhadas do esquema default de objetos que correspondem às coleções aninhadas de **$filho.**

Casos	**CriaInserção($T_{\$filho}$, $filho)**
1. Seja ei, 1<i<n, elemento multiocorrência filho de label(**$filho)** Seja T_{e_i} o tipo complexo (complex type) de $\mathbf{e_i}$ Seja $\$S_1$, $\$S_2$,...,$\$S_n$ instâncias de T_{e_i}	Valor($\$S_1$); Valor($\$S_2$);...;Valor($\$S_n$)
1. Seja ei, 1<i<n, um elemento monocorrência filho de label (**$filho)** Seja T_{e_i} o tipo $\mathbf{e_i}$ T_{e_i} possui somente um elemento(**k**) multiocorrência. Seja T_k o tipo de **k** Seja $\$S^k_1$, $\$S^k_2$,...,$\$S^k_n$ instâncias de T_k	Valor($\$S^k_1$); Valor($\$S^k_2$);...;Valor($\$S^k_n$)

Tabela 6.4: Algortimo **CriaInserção**

Casos	Valor($\$S_j$)
1.T_{e_i} tem no máximo um nível de aninhamento de valor Seja $\mathbf{VO_i}$ a VO do tipo T_{e_i}	INSERT INTO $\mathbf{VO_i}$ VALUES(**CriaObjeto1**(T_{e_i}, $\$S_j$)) + **INSERT INTO TABLE** (SELECT v.ei FROM **VO v** WHERE v.OID=<(key[6] de $T_{\$filho}$).getvalue()>) **VALUES** (SELECT REF (vo_i) FROM $\mathbf{VO_i}$ vo_i WHERE vt.OID= <(key de T_{e_i}).getvalue() >);
2.T_{e_i} tem mais de um nível de aninhamento de valor Seja $\mathbf{VO_i}$ a VO do tipo T_{e_i}	INSERT INTO **V.O** VALUES (**CriaObjeto2**(CT_{e_i}, $\$S_j$)) + **CriaInserção**($T_{\$filho}$,**$filho**) + **INSERT INTO TABLE** (SELECT v.ei FROM **VO v** WHERE v.OID=<(key de $T_{\$filho}$).getvalue()>) **VALUES** (SELECT REF (vo_i) FROM $\mathbf{VO_i}$ vo_i WHERE vt.OID= <(key de T_{e_i}).getvalue() >);

Tabela 6.5: Casos do procedimento **Valor**($\$S_j$) para **CriaInserção**

[6] Key é o elemento chave de um tipo.

Casos	Valor(S^k_j)
1.T_k tem no máximo um nível de aninhamento de valor Seja **VO_k** a VO do tipo T_k	INSERT INTO **VO_k** VALUES(**CriaObjeto**(T_k, S^k_j)) + **INSERT INTO TABLE** (SELECT v.ei FROM **VO v** WHERE v.OID=<(key de $T_{\$filho}$).getvalue()>) **VALUES** (SELECT REF (vo_k) FROM **VO_k** vo_k WHERE vt.OID= <(key de T_k).getvalue() >);
2.T_k tem mais de um nível de aninhamento de valor Seja **VO_k** a VO do tipo T_k	INSERT INTO **VO_k** VALUES(**CriaObjeto**(T_k, S^k_j)) + **CriaInserção**($T_{\$filho}$,**$filho**) + **INSERT INTO TABLE** (SELECT v.ei FROM **VO v** WHERE v.OID=<(key de $T_{\$filho}$).getvalue()>) **VALUES** (SELECT REF (vo_k) FROM **VO_k** vo_k WHERE vt.OID= <(key de T_k).getvalue() >);

Tabela 6.6: Casos do procedimento **Valor**(S^k_j) para **CriaInserção**

Observe que nos Casos 1 e 2 (Tabela 6.4), o esquema da visão XML precisa ser analisado para determinarmos as inserções geradas. Sendo assim, o Caso 1 (Tabela 6.5 e Tabela 6.6) faz chamada ao procedimento **CriaObjeto1** e o Caso 2 (Tabela 6.5 e Tabela 6.6) faz chamada aos procedimento **CriaObjeto2** e **CriaInserção**.

Caso 2:

Neste caso, uma inserção *XQueryV* é traduzida em uma inserção na coleção aninhada NT da visão de objetos VO_i, onde VO_i é a visão de objetos correspondente ao caminho $\delta_1/..\delta_i$ e $e_1^{\delta_i},.., e_p^{\delta_i}$ os elementos em δ_i que possuem um atributo correspondente em *S*. Assim, o caminho δ/label ($filho) corresponde à coleção aninhada NT da visão de objeto VO_i em *S*. Essa correspondência é obtida na geração do esquema da VOD, o qual faz um mapeamento entre os elementos do esquema da Visão XML e o esquema da VOD, como definido no Capítulo 4.

Como apresentado na Figura 6.8, o Caso 2 do algoritmo **TraduzInserçãoXQuery** é dividido em dois subcasos: Caso 2.1, o elemento label ($filho) pertence a um **TipoColeção** e Caso 2.2, o elemento label ($filho) não pertence a um **TipoColeção**. Um **TipoColeção** é um tipo complexo (ComplexType) do XML *Schema* composto por uma única definição de elemento multiocorrência.

Em ambos os casos, primeiramente cria-se o cursor TabObjPai. O cursor criado retorna um conjunto de referências dos objetos da visão de objetos VO_{n-1} (Caso 2.1 e "Senão" do Caso 2.2) ou VO_n ("Se" do Caso 2.2). Para cada referência do cursor TabObjPai, uma inserção na coleção aninhada da visão de objetos VO_{n-1} (Caso 2.1 e "Senão" do Caso 2.2) ou VO_n ("Se" do Caso 2.2) é efetuada.

Caso 2: O caminho δ/label($filho) corresponde a uma coleção aninhada **NT** em ***S***
Seja **VO_i** a VO correspondente ao caminho $\delta_1/..\delta_i$ e $e_1^{\delta_i},.., e_p^{\delta_i}$ os elementos em δ_i que possuem um atributo correspondente em ***S***

φ= φ +DECLARE **Referencia;**

Caso 2.1: O elemento label ($filho) pertence a um **TipoColeção**

φ= φ + CURSOR **TabObjPai** IS
Select REF(vo_{n-1})
From VO_1 vo_1,TABLE($vo_1 \bullet e_1^{\delta_2} \bullet .... \bullet e_p^{\delta_2}$) vo_2,.., TABLE($vo_{n-2} \bullet e_1^{\delta_{n-1}} \bullet .... \bullet e_p^{\delta_{n-1}}$) vo_{n-1}
Where $Condv_1$, $Condv_2$, $Condv_{n-1}$;

φ= φ + BEGIN OPEN **TabObjPai**

φ= φ + LOOP FETCH **TabObjPai** INTO **Referencia;**

φ= φ + EXIT WHEN **TabObjPai**%NOTFOUND;

/*Seleciona as referências dos objetos pais da coleção aninhada **NT(**$e_p^{\delta_n}$**)**da visão VO_{n-1} */

φ= φ + INSERT INTO TABLE(SELECT $vo_{n-1} \bullet e_1^{\delta_n} \bullet .... \bullet e_p^{\delta_n}$ FROM VO_{n-1} vo_{n-1} WHERE REF(vo_{n-1}) = **Referencia**) VALUES(**CriaObjeto**(**$T_{\$filho}$** , **$filho**)));

Caso 2.2: O elemento label ($filho) não pertence a um **TipoColeção**

Se o caminho δ_n possui elemento multiocorrência

/***Inserção na coleção aninhada **NT**(label($filho))da visão **$VO_n$** */

φ= φ + CURSOR **TabObjPai** IS
Select REF(vo_n) as **Referencia**
From VO_1 vo_1, TABLE($vo_1 \bullet e_1^{\delta_2} \bullet .... \bullet e_p^{\delta_2}$) vo_2,.., TABLE($vo_{n-1} \bullet e_1^{\delta_n} \bullet .... \bullet e_p^{\delta_n}$) vo_n
Where $Condv_1$, $Condv_2$, $Condv_n$

φ= φ + BEGIN OPEN **TabObjPai**

φ= φ + LOOP FETCH **TabObjPai** INTO **Referencia;**

φ= φ + EXIT WHEN **TabObjPai**%NOTFOUND;

φ = φ+ INSERT INTO TABLE(SELECT $vo_n \bullet$label($filho) FROM VO_n vo_n WHERE REF(vo_n) = **Referencia**) VALUES(CriaObjeto (**$T_{\$filho}$** , **$filho**));

Senão /* Inserção na col.aninhada **NT**($e_1^{\delta_n} \bullet .... \bullet e_p^{\delta_n} \bullet$ label($filho))da visão **$VO_{n-1}$** */

φ= φ + CURSOR **TabObjPai** IS
Select REF(vo_{n-1}) as **Referencia**
From VO_1 vo_1,TABLE($vo_1 \bullet e_1^{\delta_2} \bullet .... \bullet e_p^{\delta_2}$) vo_2,.., TABLE($vo_{n-2} \bullet e_1^{\delta_{n-1}} \bullet .... \bullet e_p^{\delta_{n-1}}$) vo_{n-1}
Where $Condv_1$, $Condv_2$, $Condv_{n-1}$

φ= φ + BEGIN OPEN **TabObjPai**

φ= φ + LOOP FETCH **TabObjPai** INTO **Referencia;**

φ= φ + EXIT WHEN **TabObjPai**%NOTFOUND;

φ= φ + INSERT INTO TABLE(SELECT $vo_{n-1} \bullet e_1^{\delta_n} \bullet .... \bullet e_p^{\delta_n} \bullet$label($filho) FROM VO_{n-1} vo_{n-1} WHERE REF(vo_{n-1}) = **Referencia**) VALUES(CriaObjeto (**$T_{\$filho}$** , **$filho**));

φ= φ + END_LOOP; END.

Figura 6.8: Caso 2 do Algoritmo **TraduzInserçãoXQuery**

Assim como no Caso 1.1, o Caso 2 também efetua uma chamada ao procedimento CriaObjeto1. Neste caso, o procedimento cria o objeto a ser inserido na coleção aninhada de uma visão de objetos VO_{n-1} ou VO_n.

Exemplo 6.3:

Considere a atualização $\mathbf{A}_2$ (Figura 6.9 (a)) que insere um elemento item para o elemento listaItens da Visão XML Pedidos_v onde o cliente é "Wamberg". A atualização *XQuery* $\mathbf{A}_2$ será traduzida na atualização SQL3 da Figura 6.9 (b). Dado que δv_1= Pedidos/Pedido, δv_2=listaItens e label($filho)=item, de acordo com os mapeamentos obtidos durante a geração do esquema da VOD, temos que o caminho $\delta v_1/\delta v_2$/label($filho) corresponde à tabela aninhada listaItens da visão de objetos **Pedidos_v**. O elemento label($filho) = item pertence ao tipo TlistaItens, o qual é composto por uma única definição de elemento multiocorrência, item. Assim, temos que TlistaItens é um TipoColeção (Caso 2.1).

Primeiramente cria-se o cursor TabObjPai (linhas 3 a 6 da Figura 6.9 (b)). O cursor criado retorna um conjunto de referências dos objetos da visão de objetos **Pedidos_v**. A expressão de caminho da cláusula FOR da linha 1 (Figura 6.9 (a)) é mapeada na cláusula FROM da linha 5 (Figura 6.9 (b)) do cursor TabObjPai. A cláusula WHERE da linha 3 (Figura 6.9 (a)) é mapeada na cláusulaWHERE da linha 6 (Figura 6.9 (b)). A cláusula SELECT da linha 4 (Figura 6.9 (b)) retorna as referências dos objetos selecionados.

Para cada referência de objeto de **Pedidos_v** retornada pelo cursor TabObjPai, uma inserção na coleção aninhada listaItens da visão de objetos **Pedidos_v** é efetuada (linhas 11 a 13 da Figura 6.9 (b)), isto é, para cada pedido onde cliente = 'Wamberg' será inserido um item na coleção aninhada listaItens da visão de objetos **Pedidos_v**. Os valores inseridos (linha 14 da Figura 6.9 (b)) são obtidos efetuando uma chamada ao procedimento **CriaObjeto1**. No Exemplo 6.2, descrevemos como obter estes valores.

Caso 3:

Neste caso, uma inserção *XQueryV* é traduzida em atualizações (UPDATE) de um atributo monovalorado de uma visão de objetos VO ou de uma coleção aninhada da visão de objetos VO, isto é, o caminho δ/label ($filho) corresponde a um atributo monovalorado em *S*. Essa correspondência é obtida na geração do esquema da VOD, o qual faz um

mapeamento entre os elementos do esquema XML e o esquema da VOD, como definido no Capítulo 4.

```
1.  for $v1 in view("Pedidos_v")/Pedidos/Pedido,
2.      $v2 in $v1/listaItens
3.  where $v1/cliente = "Wamberg"
4.  update $v2
5.  insert
6.    <item>
7.      <codigo>10</codigo>
8.      <produto>Impressora</produto>
9.      <quantidade>100</quantidade>
10.   </item>
11.
12.
13.
14.
15.
16.
```

```
DECLARE
Referencia NUMBER;
CURSOR TabObjPai IS
        SELECT REF(v1)
        FROM Pedidos_v v1
       WHERE v1.cliente = 'Wamberg'
BEGIN
OPEN TabObjPai;
LOOP
  FETCH TabObjPai INTO Referencia;
  EXIT WHEN TabObjPai %NOTFOUND;
  INSERT INTO  TABLE(  SELECT v1.listaItens
                       FROM Pedidos_v v1
                       WHERE REF(v1)= Referencia)
   VALUES(Titem('10','Impressora','100'))
END_LOOP;
END.
```

Figura 6.9: Atualização *XQueryVC* $\mathbf{A_2}$ (A) e tradução SQL3 de $\mathbf{A_2}$ (B)

Como apresentado na Figura 6.10, o Caso 3 do algoritmo **TraduzInserçãoXQuery** é dividido em dois subcasos: Caso 3.1, onde δ possui um único elemento multiocorrência **e** e Caso 3.2, onde δ possui mais de um elemento multiocorrência.

Caso 3.1:

Neste caso, dado que o caminho δ possui um único elemento multiocorrência e, uma inserção *XQueryV* é traduzida em atualizações (UPDATE) de um atributo monovalorado de uma visão de objetos VO. O atributo monovalorado da VO a ser modificado corresponde ao elemento label($filho).

Seja δv_i o caminho em δ que contém o elemento e. Caso $Condv_i$ seja "null", isto é, não exista condição para o caminho δv_i, a cláusula WHERE do comando de atualização SQL3 não é utilizada, caso contrário, a cláusula WHERE do comando de atualização SQL3 recebe $Condv_i$.

Assim como nos Casos 1.1 e 2, o Caso 3.1 também efetua uma chamada ao procedimento CriaObjeto1. Neste caso, o procedimento cria o objeto que atualiza o valor de um atributo monovalorado da visão de objetos VO.

Caso 3: O caminho δ/label ($filho) corresponde a um atributo monovalorado em ***S***

Caso 3.1: δ possui um único elem. multiocorrência **e**

Seja δv_i o caminho em δ que contém **e**.O caminho δv1/./δv_i/..**e** corresponde a uma visão de objeto **VO** em ***S***

Caso 3.1.1: $Condv_i$ é null /*Não existe condição para o caminhoδv_i*/

φ = UPDATE **VO** SET label($filho)$\bullet\delta v_{i+1}$=CriaObjeto(**$T_{\$filho}$** , **$filho**)

Caso 3.1.2: $Condv_i$ não é null /*Existe condição para o caminhoδv_i*/

φ = UPDATE **VO** SET label($filho)$\bullet\delta v_{i+1}$=CriaObjeto(**$T_{\$filho}$** , **$filho**) WHERE $Condv_i$

Caso 3.2: δ possui mais de um elemento multiocorrência

φ= φ +DECLARE **Referencia;**

Seja VO_i a visão de objetos correspondente ao caminho $\delta_1/....\delta_i$ e $e_1^{\delta_i}$, ..., $e_p^{\delta_i}$ os elementos em δ_i que possuem um atributo correspondente em ***S***

Caso 3.2.1::O caminho δ_n possui elemento multiocorrência

φ= φ + CURSOR **TabObjPai** IS

Select REF(vo_{n-1}) as **Referencia**

From VO_1 vo_1,TABLE($vo_1 \bullet e_1^{\delta_2} \bullet_{....} \bullet e_p^{\delta_2}$) vo_2,.., TABLE($vo_{n-2} \bullet e_1^{\delta_{n-1}} \bullet_{....} \bullet e_p^{\delta_{n-1}}$) vo_{n-1}

Where $Condv_1$, $Condv_2$, $Condv_{n-1}$

φ= φ + BEGIN OPEN **TabObjPai**

φ= φ + LOOP FETCH **TabObjPai** INTO **Referencia;**

φ= φ + EXIT WHEN **TabObjPai**%NOTFOUND;

Caso 3.2.1.1: $Condv_n$ é null

φ =φ + UPDATE TABLE(SELECT $vo_{n-1} \bullet e_1^{\delta_n} \bullet_{..} \bullet e_p^{\delta_n}$ FROM VO_{n-1} vo_{n-1}

WHERE REF(vo_{n-1})= **Referencia**) t

SET t.label($filho)=CriaObjeto(**$T_{\$filho}$** , **$filho**);

Caso 3.2.1.2: $Condv_n$ não é null

φ =φ+ UPDATE TABLE(SELECT $vo_{n-1} \bullet e_1^{\delta_n} \bullet_{..} \bullet e_p^{\delta_n}$ FROM VO_{n-1} vo_{n-1}

WHERE REF(vo_{n-1})= **Referencia**) t

SET t.label($filho)=CriaObjeto(**$T_{\$filho}$,$filho**)

WHERE $Condv_n$;

Caso 3.2.2: O caminho δ_n não possui elemento multiocorrência

φ= φ + CURSOR **TabObjPai** IS

Select REF(vo_{n-2}) as **Referencia**

From VO_1 vo_1,TABLE($vo_1 \bullet e_1^{\delta_2} \bullet_{....} \bullet e_p^{\delta_2}$) vo_2,.., TABLE($vo_{n-2} \bullet e_1^{\delta_{n-2}} \bullet_{....} \bullet e_p^{\delta_{n-2}}$) vo_{n-2}

Where $Condv_1$, $Condv_2$, $Condv_{n-2}$

φ= φ + BEGIN OPEN **TabObjPai**

φ= φ + LOOP FETCH **TabObjPai** INTO **Referencia;**

φ= φ + EXIT WHEN **TabObjPai**%NOTFOUND;

Caso 3.2.2.1: $Condv_{n-1}$ é null

φ =φ+ UPDATE TABLE(SELECT $vo_{n-2} \bullet e_1^{\delta_{n-1}} \bullet_{..} \bullet e_p^{\delta_{n-1}}$ FROM VO_{n-2} vo_{n-2}

WHERE REF(vo_{n-2})= **Referencia**) t

SET t.δ_n.label($filho)=CriaObjeto(**$T_{\$filho}$** , **$filho**);

Caso 3.2.1: $Condv_{n-1}$ não é null

φ =φ+ UPDATE TABLE(SELECT $vo_{n-2} \bullet e_1^{\delta_{n-1}} \bullet_{..} \bullet e_p^{\delta_{n-1}}$ FROM VO_{n-2} vo_{n-2}

WHERE REF(vo_{n-2})= **Referencia**) t

SET t.δ_n.label($filho)=CriaObjeto(**$T_{\$filho}$** , **$filho**) WHERE $Condv_{n-1}$;

φ= φ + END_LOOP; END.

Figura 6.10: Caso 3 do Algoritmo **TraduzInserçãoXQuery**

Caso 3.2:

Neste caso, dado que o caminho δ possui mais de um elemento multiocorrência. Uma inserção *XQueryV* é traduzida em atualizações (UPDATE) de um atributo monovalorado de uma coleção aninhada da visão de objetos VO_{n-1} (Caso 3.2.1) ou VO_{n-2} (Caso 3.2.2). O atributo monovalorado a ser modificado corresponde ao elemento label($filho).

Em ambos os casos (Caso 3.2.1 e Caso 3.3.2), primeiramente cria-se o cursor TabObjPai. O cursor criado retorna um conjunto de referências dos objetos da visão de objetos VO_{n-1} (Caso 3.2.1) ou VO_{n-2} (Caso 3.2.2). Para cada referência do cursor TabObjPai, uma modificação do atributo da coleção aninhada da visão de objetos VO_{n-1} (Caso 3.2.1) ou VO_n (Caso 3.2.2) é efetuada. Os Casos 3.2.1 e 3.2.2 são subdivididos para considerar as situações onde existe ou não uma condição.

Assim como no Caso 3.1, o Caso 3.2 também efetua uma chamada ao procedimento CriaObjeto1. Neste caso, o procedimento cria o objeto que atualiza o valor de um atributo monovalorado da coleção aninhada da visão de objetos VO_{n-1} ou VO_{n-2}.

Exemplo 6.4:

Considere a atualização **A**$_3$ (Figura 6.11 (a)) que insere um elemento quantidade para o elemento item na Visão XML Pedidos_v. A atualização *XQuery* **A**$_3$ será traduzida na atualização SQL3 da Figura 6.11 (b). Dado que δv_1= Pedidos/Pedido, δv_2=listaItens/item e label($filho)=quantidade, de acordo com os mapeamentos obtidos durante a geração do esquema da VOD, temos que o caminho $\delta v_1/\delta v_2$/label($filho) corresponde ao atributo monovalorado quantidade da coleção aninhada **listaItens** da visão de objetos **Pedidos_v**.

Como δ possui mais de um elemento multiocorrência, Pedido e item, o caminho δv_2 possui um elemento multiocorrência, e $Cond v_2$ não é nulo. Desta forma, o Caso 3.2.1.2 do Algoritmo **TraduzInserçãoXQuery** deve ser aplicado.

Primeiramente, cria-se o cursor TabObjPai (linhas 3 a 6 da Figura 6.11 (b)). O cursor criado retorna um conjunto de referências dos objetos da visão de objetos **Pedidos_v**. A expressão de caminho da cláusula FOR da linha 1 (Figura 6.11 (a)) é mapeada na cláusula FROM da linha 5 (Figura 6.11 (b)) do cursor TabObjPai. A cláusula WHERE da linha 3

(Figura 6.11 (a)) é mapeada na cláusula WHERE da linha 6 (Figura 6.11 (b)). A cláusula SELECT da linha 4 (Figura 6.11 (b)) retorna as referências dos objetos selecionados.

Para cada referência de objeto da visão **Pedidos_v** retornada pelo cursor TabObjPai, uma modificação do atributo quantidade da coleção aninhada listaItens da visão de objetos **Pedidos_v** é efetuada (linhas 11 a 15 da Figura 6.11 (b)), isto é, para cada pedido onde cliente = 'Wamberg', é atribuído ao atributo quantidade da coleção aninhada listaItens da visão de objetos **Pedidos_v** o valor "100", onde o código é "10". O valor atribuído ao atributo quantidade (linha 14 da Figura 6.11 (b)) é obtido efetuando uma chamada ao **CriaObjeto1**. No Exemplo 6.1, descrevemos como obter este valor.

```
1.  for $v1 in view("Pedidos_v")/Pedidos/Pedido,
2.      $v2 in $v1/listaItens/item
3.  where $v1/cliente = "Wamberg" AND
4.          $v2/codigo = 10
5.  update $v2
6.  insert
7.     <quantidade>100</quantidade>
8.
9.
10.
11.
12.
13.
14.
15.
16.
```

```
DECLARE
Referencia NUMBER;
CURSOR TabObjPai IS
        SELECT REF(v1)
        FROM Pedidos_v v1
       WHERE v1.cliente = 'Wamberg'
BEGIN
OPEN TabObjPai;
LOOP
  FETCH TabObjPai INTO Referencia;
  EXIT WHEN TabObjPai %NOTFOUND;
  UPDATE TABLE(  SELECT v1.listaItens
                 FROM Pedidos_v v1
                 WHERE REF(v1)=Referencia) v2
   SET v2.quantidade =100
  WHERE v2.codigo =10
END_LOOP;
END.
```

Figura 6.11: Atualização *XQueryVC* $\mathbf{A_3}$ (A) e tradução SQL3 de $\mathbf{A_3}$ (B)

6.3.2 Algoritmo para Traduzir uma Remoção *XQueryV* em Remoções ou Modificações em SQL3

Nesta seção apresentamos o algoritmo implementado no módulo Tradutor do *XUpdateTranslator* para traduzir remoções *XQueryV* de uma visão XML V_x, em atualizações SQL3 sobre a respectiva visão de objetos *default* VO, a partir do esquema de V_x.

Como apresentado na Figura 6.12, o algoritmo **TraduzRemoçãoXQuery** recebe como entrada uma atualização *XQueryV* e o XML *Schema* da respectiva visão XML. O algoritmo **TraduzRemoçãoXQuery** traduz remoções *XQueryV* de uma visão XML V_x, em atualizações

SQL3 sobre a respectiva visão de objetos *default* VO para os dois casos apresentados a seguir.

Algoritmo TraduzRemoçãoXQuery (**XQ**: Expressão *XqueryV*)
φ: **string**;
Início
φ =∅;
Seja **δ** = $\delta v_1/\delta v_2/\ldots./\delta v_n$ o caminho da raiz até δv_n
Seja ***S*** o esquema default de objetos
Caso 1: O caminho δ corresponde a uma visão de objeto **VO** em ***S***

Caso 2: O caminho δ corresponde a uma tabela aninhada **NT** em ***S***

Figura 6.12: Algoritmo **TraduzRemoçãoXQuery**

Caso 1:

Neste caso, o caminho δ corresponde a uma visão de objeto VO em ***S***. Esta correspondência é obtida na geração do esquema da VOD, a qual faz um mapeamento entre os elementos do esquema de V_x e o esquema da VOD VO, como definido no Capítulo 4. Como apresentado na Figura 6.13, o Caso 1 do algoritmo **TraduzRemoçãoXQuery** é dividido em dois subcasos.

Caso 1: O caminho δ corresponde a uma visão de objeto **VO** em ***S***
Caso 1.1: label($filho) <> **δ**
Caso 1.1.1: $Condv_1$ é null
φ = UPDATE **VO** SET label($filho) = NULL;
Caso 1.1.2: $Condv_1$ não é null
φ = UPDATE **VO** SET label($filho) = NULL WHERE $Condv_1$;
Caso 1.2: label($filho) = **δ**
Caso 1.2.1: $Condv_1$ é null
φ = DELETE FROM **VO;**
Caso 1.2.1: $Condv_1$ nãoé null
φ = DELETE FROM **VO** WHERE $Condv_1$;

Figura 6.13: Caso 1 do Algoritmo **TraduzRemoçãoXQuery**

Caso 1.1:

No Caso 1.1, label($filho) é diferente de δ, isto é, label($filho) corresponde a um atributo da visão de objeto VO em ***S***. Assim, uma remoção *XQueryV* de uma visão XML V_x é traduzida em uma modificação (UPDATE) SQL3 do atributo da VO que corresponde a label($filho). Caso $Condv_1$ seja nula, a cláusula WHERE da modificação SQL3 não é utilizada.

Exemplo 6.5:

Considere a atualização $\mathbf{A_4}$ (Figura 6.14 (a)) que remove o elemento dataentrega na Visão XML Pedidos_v onde o cliente é "Denis". A atualização *XQuery* $\mathbf{A_4}$ será traduzida na atualização SQL3 da Figura 6.14 (b). De acordo com os mapeamentos obtidos durante a geração do esquema da VOD, o caminho $\boldsymbol{\delta v_1}$ =Pedidos/Pedido corresponde à visão de objetos **Pedidos_v**. Dado que $\boldsymbol{\delta v_1}$ está associado a variável $\mathbf{\$v_1}$, $Condv_1$ não é nula, e label($filho) = dataentrega, que é diferente de $\boldsymbol{\delta v_1}$, então $\mathbf{A_4}$ é traduzida em uma modificação do valor do atibuto dataentrega da visão de objetos **Pedidos_v** onde cliente é "Denis".

A expressão de caminho da cláusula for da linha 1 (Figura 6.14 (a)) é mapeada na cláusula UPDATE da linha 1 (Figura 6.14 (b)). A cláusula WHERE da linha 2 (Figura 6.14 (a)) é mapeada na cláusula WHERE da linha 3 (Figura 6.14 (b)). A cláusula SET (Linha 2 da Figura 6.14 (a)) atribui ao atributo dataentrega o valor NULL.

1.	For $v1 in view("Pedidos_v")/Pedidos/Pedido	UPDATE Pedidos v1
2.	where $v1/cliente = "Denis"	SET v1.dataentrega = NULL
3.	update $v1	WHERE v1.cliente = 'Denis'
4.	delete $v1/dataentrega **(A)**	**(B)**
5.		
6.		

Figura 6.14: Atualização *XQueryVC* $\mathbf{A_4}$ (A) e tradução SQL3 de $\mathbf{A_4}$ (B)

Caso 1.2:

No Caso 1.2, label($filho) é o próprio caminho δ, isto é, label($filho) corresponde a a visão de objetos VO. Assim, uma remoção *XQueryV* de uma visão XML V_x é traduzida em uma remoção SQL3 em VO. Assim como no Caso 1.1, caso $Condv_1$ seja nula, a cláusula WHERE da modificação SQL3 não é utilizada.

Exemplo 6.6:

Considere a atualização $\mathbf{A_5}$ (Figura 6.15 (a)) que remove o elemento pedido na Visão XML Pedidos_v onde o cliente é "Denis". A atualização *XQuery* $\mathbf{A_5}$ será traduzida na atualização SQL3 da Figura 6.15 (b). De acordo com os mapeamentos obtidos durante a geração do esquema da VOD, o caminho $\boldsymbol{\delta v_1}$ =Pedidos/Pedido corresponde à visão de

objetos **Pedidos_v**. Dado que δv_1 está associado a variável **$v1** , $Condv_1$ não é nulo, e label($filho) = **$v1,** então A_5 é traduzida em uma remoção na visão de objetos **Pedidos_v** onde cliente é "Denis".

A expressão de caminho da cláusula for da linha 1 (Figura 6.15 (a)) é mapeada na cláusula DELETE da linha 1 (Figura 6.15 (b)). A cláusula WHERE da linha 2 (Figura 6.15 (a)) é mapeada na cláusula WHERE da linha 2 (Figura 6.15 (b)).

1.	for $v1 in view("Pedidos_v")/Pedidos/Pedido	DELETE FROM Pedidos_v v1
2.	where $v1/cliente = "Denis"	WHERE v1.cliente = 'Denis'
3.	update $v1	
4.	delete $v1 **(A)**	**(B)**
5.		
6.		

Figura 6.15: Atualização *XQueryVC* **A5** (A) e tradução SQL3 de **A5** (B)

Caso 2:

Neste caso, o caminho δ corresponde a uma coleção aninhada NT da visão de objeto VO_i, onde VO_i é a visão de objetos correspondente ao caminho $\delta_1/..\delta_i$ e $e_1^{\delta_i},.., e_p^{\delta_i}$ os elementos em δ_i que possuem um atributo correspondente em ***S***. Como apresentado na Figura 6.16, o Caso 2 do algoritmo **TraduzRemoçãoXQuery** é dividido em dois subcasos.

Em ambos os casos, primeiramente cria-se o cursor TabObjPai. O cursor criado retorna um conjunto de referências dos objetos da visão de objetos VO_{n-1}, onde VO_{n-1} é a visão de objetos correspondente ao caminho $\delta_1/..\delta_{n-1}$. Para cada referência do cursor TabObjPai, uma atualização SLQ3 (UDPATE ou DELETE) é efetuada.

Caso 2.1:

No Caso 2.1, label($filho) é diferente de δ, isto é, label($filho) corresponde a um atributo da coleção aninhada NT da visão de objeto VO_{n-1} em ***S***. Assim, uma remoção *XQueryV* de uma visão XML V_x é traduzida em uma atualizações (UPDATE) de um atributo da coleção aninhada NT da visão de objetos VO_{n-1}. O atributo monovalorado a ser modificado corresponde ao elemento label($filho). A modificação ocorre na coleção aninhada NT da visão de objetos VO_{n-1}, pois label($filho) pertence ao tipo do o elemento

$e_p^{\delta_n}$,do caminho δ_n. Caso a condição $Condv_n$ seja nula, a cláusula WHERE da modificação SQL3 não é utilizada.

Caso 2: O caminho δ corresponde a uma coleção aninhada **NT** em ***S***
Seja $\mathbf{VO_i}$ a visão de objetos correspondente ao caminho $\delta_1/....\delta_i$ e $e_1^{\delta_i}$, ..., $e_p^{\delta_i}$ os elementos em δ_i que possui um atributo correspondente em ***S***

φ= φ +DECLARE **Referencia;**
φ= φ + CURSOR **TabObjPai** IS
Select REF(vo_{n-1})
From VO_{n-1} vo_{n-1}
Where $Condv_{n-1}$
φ= φ + BEGIN OPEN **TabObjPai**
φ= φ + LOOP FETCH **TabObjPai** INTO **Referencia;**
φ= φ + EXIT WHEN **TabObjPai**%NOTFOUND;

Caso 2.1: label($filho) <> **δ**
Caso 2.1.1: $Condv_n$ não é null
φ= φ + UPDATE TABLE(SELECT $vo_{n-1}.e_1^{\delta_n}.....e_p^{\delta_n}$ FROM VO_{n-1} vo_{n-1}
WHERE REF(vo_{n-1}) = **Referencia**)
SET label($filho) = NULL
WHERE $Condv_n$;
Caso 2.1.2: $Condv_n$ é null
φ= φ + UPDATE TABLE(SELECT $vo_{n-1}.e_1^{\delta_n}.....e_p^{\delta_n}$ FROM VO_{n-1} vo_{n-1}
WHERE REF(vo_{n-1}) = **Referencia**)
SET label($filho) = NULL;
Caso 2.2: label($filho) = **δ**
Caso 2.2.1: $Condv_n$ não é null
φ= φ + DELETE FROM TABLE(SELECT $vo_{n-1}.e_1^{\delta_n}.....e_p^{\delta_n}$ FROM VO_{n-1}
vo_{n-1} WHERE REF(vo_{n-1}) = **Referencia**)
WHERE $Condv_n$;

Caso 2.2.2: $Condv_n$ é null
φ= φ + DELETE FROM TABLE(SELECT $vo_{n-1}.e_1^{\delta_n}.....e_p^{\delta_n}$ FROM VO_{n-1}
vo_{n-1} WHERE REF(vo_{n-1}) = **Referencia**);

φ= φ + END_LOOP; END.

Figura 6.16: Caso 2 do Algoritmo TraduzRemoçãoXQuery

Caso 2.2:

No Caso 2.2, label($filho) é o próprio caminho δ, isto é, label($filho) corresponde a coleção aninhada NT da a visão de objetos VO_{n-1}. Assim, uma remoção *XQueryV* de uma visão XML V_x é traduzida em uma remoção SQL3 na coleção aninhada NT da visão de objetos VO_{n-1}. A remoção ocorre na coleção aninhada NT da visão VO_{n-1}, pois label($filho)

pertence ao tipo do o elemento $e_p^{\delta_n}$. Assim como no Caso 2.1, caso $Condv_n$ seja nula, a cláusula WHERE do comando de modificação SQL3 não é utilizada.

Exemplo 6.7:

Considere a atualização $\mathbf{A_6}$ (Figura 6.17 (a)) que remove o elemento item do elemento listaItens na Visão XML Pedidos_v onde o cliente do Pedido é "Denis" e o produto é "Mouse". A atualização *XQuery* $\mathbf{A_6}$ será traduzida na atualização SQL3 da Figura 6.17(b). O caminho $\boldsymbol{\delta = \delta v_1/\delta v_2}$, onde $\boldsymbol{\delta v_1}$=Pedidos/Pedido e $\boldsymbol{\delta v_2}$=listaItens/item corresponde à coleção aninhada **listaItens** da visão de objetos **Pedidos_v**, de acordo com os mapeamentos obtidos durante a geração do esquema da VOD**.** Dado que $\boldsymbol{\delta v_2}$ está associado a variável $\mathbf{\$v_2}$, $Condv_2$ não é nulo, e label($filho) = $\mathbf{\$v_2}$, $\mathbf{A_6}$ é traduzida em uma remoção na coleção aninhada **listaItens** da visão de objetos **Pedidos_v**. Desta forma, o Caso 2.2.1 do Algoritmo **TraduzRemoçãoXQuery** deve ser aplicado.

Primeiramente cria-se o cursor TabObjPai (linhas 3 a 6 da Figura 6.17 (b)). O cursor criado retorna um conjunto de referências dos objetos da visão de objetos **Pedidos_v**. A expressão de caminho da cláusula FOR da linha 1 (Figura 6.17 (a)) é mapeada na cláusula FROM da linha 5 (Figura 6.17 (b)) do cursor TabObjPai. A cláusula WHERE da linha 3 (Figura 6.17 (a)) é mapeada na cláusula WHERE da linha 6 (Figura 6.17 (b)). A cláusula SELECT da linha 4 (Figura 6.17 (b)) retorna as referências dos objetos selecionados.

Para cada referência de objeto de Pedidos_v retornada pelo cursor TabObjPai, uma remoção na coleção aninhada listaItens do objeto referenciado (linhas 12 a 15 da Figura 6.17 (b)) é efetuada, isto é, para cada pedido do cliente "Wamberg", remover os itens da coleção listaItens onde produto é "Mouse". A cláusula WHERE da linha 4 (Figura 6.17 (a)) é mapeada na cláusula WHERE da linha 15 (Figura 6.17 (b)).`

1.	For $v1 in view("Pedidos_v")/Pedidos/	DECLARE
2.	Pedido, $v2 in $v1/listaItens/item	Referencia NUMBER;
3.	where $v1/cliente = "Wamberg" and	CURSOR TabObjPai IS
4.	$v2/produto = "Mouse"	SELECT REF(v1)
5.	update $v2	FROM Pedidos_v v1
6	delete $v2	WHERE v1.cliente = 'Wamberg'
7.		BEGIN
8.		OPEN TabObjPai;
9.		LOOP
10.		FETCH TabObjPai INTO Referencia;
11.		EXIT WHEN TabObjPai %NOTFOUND;
12.		DELETE FROM TABLE
13.		(SELECT v1.listaItens FROM Pedidos_v v1
14.		WHERE REF(v1)= Referencia) v2
15.		WHERE v2.produto = 'Mouse'
16.	**(A)**	END_LOOP;
17.		**END.** **(B)**

Figura 6.17: Atualização *XQueryVC* $\mathbf{A_6}$ (A) e tradução SQL3 de $\mathbf{A_6}$ (B)

6.3.3 Algoritmo para Traduzir uma Modificação *XQueryV* em Modificações SQL3

Nesta seção apresentamos o algoritmo implementado no módulo Tradutor do *XQueryTranslator* para transformar modificações *XQueryV* de uma visão XML V_x, em atualizações SQL3 sobre a respectiva visão de objetos *default* VO, a partir do esquema de V_x.

Como apresentado na Figura 6.18, o algoritmo **TraduzModificaçãoXQuery** recebe como entrada uma atualização *XQueryV* e o XML *Schema* da respectiva visão XML. O algoritmo **TraduzModificaçãoXQuery** traduz modificações *XQueryV* de uma visão XML V_x, em atualizações SQL3 sobre a respectiva visão de objetos *default* VO para os dois casos apresentados a seguir:

Algoritmo TraduzModificaçãoXQuery (**XQ**: Expressão *XQueryV*)
φ: **string**;
Início
φ =∅;
Seja **δ** = $\delta v_1/\delta v_2/.../\delta v_n$ o caminho da raiz até δv_n
Seja ***S*** o esquema default de objetos
Caso 1: O caminho δ corresponde a uma visão de objeto **VO** em ***S***

Caso 2: O caminho δ corresponde a uma tabela aninhada **NT** em ***S***

Figura 6.18: Algoritmo **TraduzModificaçãoXQuery**

Caso 1:

Neste caso, o caminho δ corresponde a uma visão de objeto VO em ***S***. Esta correspondência é obtida na geração do esquema da VOD, o qual faz um mapeamento entre os elementos do esquema de V_x e o esquema da da VOD VO, como definido no capítulo 4. Como apresentado na Figura 6.19, o Caso 1 do algoritmo **TraduzModificaçãoXQuery** é dividido em dois subcasos.

Caso 1.1:

No Caso 1.1, label($filho) é uma referência para um elemento do tipo T'. Seja **VO'** a visão de objetos associada a T' e **OID,** o identificador único da visão **VO'**. Então, uma modificação *XQueryV* de uma visão XML V_x é traduzida em uma modificação (UPDATE) SQL3 do atributo da **VO** que corresponde a label($filho). O atributo modificado recebe a referência de um objeto do tipo T'. Caso $Condv_1$ seja nula, a cláusula WHERE da modificação SQL3 não é utilizada.

Caso 1: O caminho δ corresponde a uma visão de objeto **VO** em ***S***
Caso 1.1: $filho é uma referência para um elemento do tipo T'
Seja **VO'** a visão de objetos associada a T' e **OID,** o identificador único da visão **VO'**.
Caso 1.1.1: $Condv_1$ não é null

```
φ =   UPDATE VO
      SET label($filho) = (Select REF(vo') From VO' vo' Where vo'.OID =
      label($conteudo).getvalue())
      WHERE Condv1;
```

Caso 1.1.2: $Condv_1$ é null

```
φ =   UPDATE VO
      SET label($filho) = (Select REF(vo') From VO' vo' Where vo'.OID =
      label($conteudo).getvalue());
```

Caso 1.2: $filho não é uma referência para um elemento do tipo T'
Caso 1.2.1: $Condv_1$ não é null

```
φ =   UPDATE VO
      SET label($filho) =  label($conteudo).getvalue()
      WHERE Condv1;
```

Caso 1.2.2: $Condv_1$ é null

```
φ =   UPDATE VO
      SET label($filho) =  label($conteudo).getvalue();
```

Figura 6.19: Caso 1 do Algoritmo **TraduzModificaçãoXQuery**

Caso 1.2:

No Caso 1.2, label($filho) não é uma referência para um elemento do tipo T'. Então, uma modificação *XQueryV* de uma visão XML V_x é traduzida em uma modificação (UPDATE) SQL3 do atributo da **VO** que corresponde a label($filho). O atributo modificado

recebe o valor de $conteúdo, o qual é obtido a partir do método getvalue(). Caso $Condv_1$ seja nula, a cláusula WHERE do comando de modificação SQL3 não é utilizada.

Exemplo 6.8:

Considere a atualização $\mathbf{A_7}$ (Figura 6.20 (a)) que modifica o valor do elemento cliente para "Denis Roberto" na Visão XML Pedidos_v, onde o cliente é "Denis". A atualização *XQuery* $\mathbf{A_7}$ será traduzida na atualização SQL3 da Figura 6.20(b). De acordo com os mapeamentos obtidos durante a geração do esquema da VOD, o caminho δ =Pedidos/Pedido corresponde à visão de objetos **Pedidos_v**. Dado que o tipo do elemento cliente não é de referëncia e $Condv_1$ não é nulo**,** $\mathbf{A_7}$ é traduzida em uma atualização do atributo cliente na visão de objetos **Pedidos_v** (Caso 1.2.1).

A expressão de caminho da cláusula for da linha 1 (Figura 6.20 (a)) é mapeada na cláusula UPDATE da linha 1 (Figura 6.20 (b)). A cláusula WHERE da linha 2 (Figura 6.20 (a)) é mapeada na cláusula WHERE da linha 3 (Figura 6.20 (b)). A cláusula REPLACE (Linha 4 da Figura 6.20 (a)) é mapeada na cláusula SET (Linha 2 da Figura 6.20 (b)).

1.	for $v1 in view("Pedidos_v")/Pedidos/Pedido	UPDATE Pedidos_v v1
2.	where $v1/cliente = "Denis"	SET v1.cliente = 'Denis Roberto'
3.	update $v1	WHERE v1.cliente = 'Denis'
4.	replace $v1/cliente with <cliente>Denis Roberto</cliente>	**(B)**
5.		
6.	**(A)**	

Figura 6.20: Atualização *XQueryVC* $\mathbf{A_7}$ (A) e tradução SQL3 de $\mathbf{A_7}$ (B)

Caso 2:

Neste caso, o caminho δ corresponde à coleção aninhada NT da visão de objeto VO_i, onde VO_i é a visão de objetos correspondente ao caminho $\delta_1/..\delta_i$ e $e_1^{\delta_i},.., e_p^{\delta_i}$ os elementos em δ_i que possui um atributo correspondente em ***S***. Então, uma modificação *XQueryV* de uma visão XML V_x é traduzida em uma modificação (UPDATE) SQL3 de um atributo da coleção aninhada NT da $\mathbf{VO_{n-1}}$. O atributo modificado corresponde ao elemento label($filho), este recebe o valor de $conteúdo, o qual é obtido a partir do método getvalue(). Caso $Condv_n$ seja nula, a cláusula WHERE do comando de modificação SQL3 não é

utilizada. A modificação ocorre na coleção aninhada NT da visão de objetos VO_{n-1}, pois label($filho) pertence ao tipo do o elemento $e_p^{\delta_n}$,do caminho δ_n

Como apresentado na Figura 6.21, primeiramente cria-se o cursor TabObjPai. O cursor criado retorna um conjunto de referências dos objetos da visão de objetos VO_{n-1}. Para cada referência do cursor TabObjPai, uma modificação SLQ3 (UDPATE) é efetuada.

EXEMPLO 6.9:

Considere a atualização $\mathbf{A_8}$(Figura 6.22(a)) que modifica o valor do elemento produto do elemento item na Visão XML Pedidos_v para "Mouse", onde o cliente é "Wamberg" e a quantidade "3". A atualização *XQuery* $\mathbf{A_8}$ será traduzida no comando de atualização SQL3 da Figura 6.22(b). O caminho $\delta = \delta v_1/\delta v_2$, onde δv_1=Pedidos/Pedido e δv_2=listaItens/item, corresponde à coleção aninhada **listaItens** da visão de objetos **Pedidos_v**, de acordo com os mapeamentos obtidos durante a geração do esquema da VOD. Assim, $\mathbf{Q_8}$ é traduzida em uma modificação do atributo produto da coleção aninhada **listaItens** da visão de objetos **Pedidos_v**.

Caso 2: O caminho δ corresponde a uma coleção aninhada **NT** em ***S***
Seja $\mathbf{VO_i}$ a visão de objetos correspondente ao caminho $\delta_1/....\delta_i$ e $e_1^{\delta_i}$, ..., $e_p^{\delta_i}$ os elementos em δ_i que possui um atributo correspondente em ***S***

φ= φ +DECLARE **Referencia;**
φ= φ + CURSOR **TabObjPai** IS
Select REF(vo_{n-1}) as **Referencia**
From VO_{n-1} vo_{n-1}
Where $Condv_{n-1}$

φ= φ + BEGIN OPEN **TabObjPai**
φ= φ + LOOP FETCH **TabObjPai** INTO **Referencia;**
φ= φ + EXIT WHEN **TabObjPai**%NOTFOUND;
Caso 2.1: $Condv_n$ não é null
φ= φ +
UPDATE TABLE(SELECT $vo_{n-1}.e_1^{\delta_n}.....e_p^{\delta_n}$ FROM VO_{n-1} vo_{n-1}
WHERE REF(vo_{n-1}) = **Referencia**)
SET label($filho) = label($filho).getvalue()
WHERE $Condv_n$;
Caso 2.2: $Condv_n$ é null
φ= φ +
UPDATE TABLE(SELECT $vo_{n-1}.e_1^{\delta_n}.....e_p^{\delta_n}$ FROM VO_{n-1} vo_{n-1}
WHERE REF(vo_{n-1}) = **Referencia**)
SET label($filho)= label($filho).getvalue();
φ= φ + END_LOOP; END.

Figura 6.21: Caso 2 do Algoritmo **TraduzModificaçãoXQuery**

Primeiramente cria-se o cursor TabObjPai (linhas 3 a 6 da Figura 6.22 (b)). O cursor criado retorna um conjunto de referências dos objetos da visão de objetos **Pedidos_v**. A expressão de caminho da cláusula FOR da linha 1 (Figura 6.22 (a)) é mapeada na cláusula FROM da linha 5 (Figura 6.22 (b)) do cursor TabObjPai. A cláusula WHERE da linha 3 (Figura 6.22 (a)) é mapeada na cláusula WHERE da linha 6 (Figura 6.22 (b)). A cláusula SELECT da linha 4 (Figura 6.22 (b)) retorna as referências dos objetos selecionados.

Para cada referência de objeto de Pedidos_v retornada pelo cursor TabObjPai, uma modificação do atributo produto na tabela aninhada listaItens do objeto referenciado (linhas 12 a 16 da Figura 6.14 (b)) é efetuada, isto é, para cada pedido do cliente "Wamberg", atribuir "Mouse" ao atributo produto onde a quantidade seja 3. A cláusula replace (Linha 7 da Figura 6.22 (a)) é mapeada na cláusula SET (Linha 15 da Figura 6.22 (b)). A cláusula *WHERE* da linha 4 (Figura 6.22 (a)) é mapeada na cláusula WHERE da linha 16 (Figura 6.22 (b)).

1.	For $v1 in view("Pedidos_v")/Pedidos/	DECLARE
2.	Pedido, $v2 in $v1/listaItens/item	Referencia NUMBER;
3.	where $v1/cliente = "Wamberg" and	CURSOR TabObjPai IS
4.	$v2/quantidade = 3	SELECT REF(v1)
5.	update $v2	FROM Pedidos_v v1
6	replace $v2/produto with	WHERE v1.cliente = 'Wamberg'
7.	<produto>Mouse</produto>	BEGIN
8.		OPEN TabObjPai;
9.		LOOP
10.		FETCH TabObjPai INTO Referencia;
11.		EXIT WHEN TabObjPai %NOTFOUND;
12.		UPDATE TABLE
13.		(SELECT v1.listaItens FROM Pedidos_v v1
14.		WHERE REF(v1)= Referencia) v2
15.		SET v2.produto = 'Mouse'
16.		WHERE v2.quantidade = 3
17.	**(A)**	END_LOOP;
		END. **(B)**

Figura 6.22: Atualização *XQueryVC* $\mathbf{A_8}$ (A) e tradução SQL3 de $\mathbf{A_8}$ (B)

Capítulo 7 - Conclusões

Neste trabalho abordamos o problema de atualização de bancos de dados objeto-relacionais através de visões XML. No nosso enfoque, usamos visões de objeto *default* (VOD) como interface entre visões XML e o banco de dados. Uma VOD é uma visão de objetos que possui esquema compatível com o esquema da visão XML, isto é, possui os mesmos tipos e a mesma estrutura do esquema da visão XML. Assim, a tradução de atualizações definidas sobre a visão XML em atualizações sobre a VOD é facilitada.

No enfoque proposto, utilizamos o XML *Publisher*, um *framework* para publicar dados objeto-relacionais derivados do *Oracle* 9i na *web*, através de visões XML. Para publicar uma visão XML no *XML Publisher*, o projetista deve definir uma VOD através de uma consulta SQL3 que especifica como os objetos da visão são sintetizados a partir dos dados do banco de dados e definir os tradutores de atualização da VOD através de um bloco de comandos PL/SQL3 que especificam como atualizações definidas sobre a visão de objeto são traduzidas em atualizações especificadas sobre o banco de dados. Em seguida, a visão XML pode ser atualizada usando uma extensão da linguagem *XQuery* definida em [18]. XML *Publisher* define o seguinte conjunto de operações de acesso: *GetCapabilities*, *GetSchema*, *ExecuteQuery* e *ExecuteUpdate*.

A execução das atualizações *XQuery* no XML *Publisher* é feita no *XUpdateTranslator*, onde atualizações *XQuery* são traduzidas em atualizações SQL3 na VOD. A tradução é feita com base no XML *Schema* da visão XML.

Outra contribuição deste trabalho foi o TAV (Tradutor de Atualização de Visão), um protótipo de uma ferramenta para auxiliar a tarefa da geração dos tradutores (*instead of triggers*) de atualização da VOD. A novidade do enfoque proposto é que as visões de objetos são especificadas através de assertivas de correspondência que especificam formalmente o relacionamento entre o esquema da visão e o esquema do banco de dados. O formalismo permite especificar várias formas de correspondência, inclusive casos onde os esquemas possuem estruturas diferentes. A ferramenta oferece uma interface gráfica para apoiar a criação do tipo da visão e a edição de suas assertivas de correspondência, e gera, de forma automática, os tradutores de atualização que realizam o mapeamento definido pelas assertivas da visão. Os algoritmos apresentados no Capítulo 5 recebem como entrada o

esquema da visão, o esquema do banco de dados e as assertivas da visão de objetos, e geram, de forma automática, os tradutores de atualização. A vantagem do formalismo utilizado é que permite provar formalmente que os tradutores gerados pelos algoritmos traduzem corretamente as atualizações.

Desenvolvemos ainda um ambiente que dá suporte ao processo de publicação de visões XML no XML *Publisher*. O processo de publicação compreende os seguintes passos: (i) o projetista define, através de uma interface gráfica, o tipo dos elementos da visão XML; (ii) a ferramenta CriaEsquemaVOD gera automaticamente o esquema da VOD; (iii) o projetista deve então carregar o esquema do banco e, através de uma GUI, definir as assertivas de correspondência do esquema VOD com o esquema do banco de dados; e (iv) com base no conjunto de assertivas de correspondência da visão, a ferramenta TAV gera automaticamente os tradutores (*instead of triggers*) de atualização da visão de acordo o mapeamento definido pelas assertivas.

Como trabalhos futuros pretendemos:

- Estender o *XUpdateTranslator* para traduzir atualizações XQuery em atualizações definidas diretamente sobre as tabelas do banco de dados. Assim, o *XUpdateTranslator* poderá ser usado também para bancos de dados que não oferecem o mecanismo de visões de objetos;
- Estender o *XUpdateTranslator* para suportar um conjunto maior de atualizações *XQuery*;
- Estender o TAV para:
    - Gerar tradutores de atualização para outros tipos de visão de objeto, i.e; definidas com novas formas de assertivas. Neste trabalho consideramos apenas as visões que preservam objetos;
    - Gerar tradutores de visões de objetos para bancos de dados objeto-relacionais.

ANEXO A

Neste Anexo mostramos que os tradutores de atualização gerados pelos algoritmos apresentados no Capítulo 5 são corretos; i.é; traduzem corretamente a atualização na visão solicitada. Nas seções seguintes, seja:

- *U* a lista de atualizações requisitadas pelo tradutor (U= $\tau(\mu)$);
- σ_s e σ'_s são os estados de **S** imediatamente antes e depois de U;
- σ_v e σ'_v os estados da visão **V** em σ_s e σ'_s;
- σ''_v é o novo estado de V após a atualização solicitada μ..

A.1: Tradutores de Inserção em Coleção Aninhada

A.1.1: Caso 1- Ligação multivalorada é a última

A seguir mostramos que o tradutor AdiçãoEmLista_T_c_Caso1 da figura 5.10 realiza a atualização μ solicitada na visão; i.é, a adição do objeto :new do tipo $\mathbf{T_c}$ na coleção aninhada lista_T_c do objeto :parent da visão **V** de tipo $\mathbf{T_v}$.

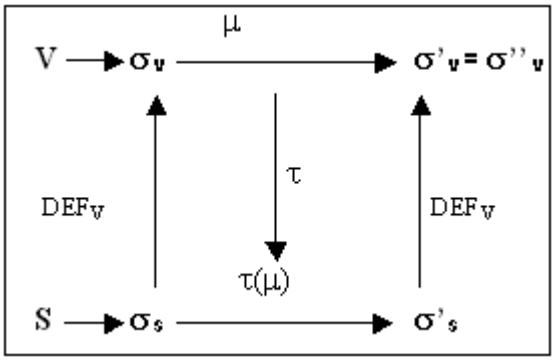


Figura A.1: Tradução de Atualização de Visão

- Temos que provar que:

(1) σ'_v(:parent•**lista_T_c**) = σ''_v(:parent•**lista_T_c**). (vide Figura A.1).

De μ temos que:

(2) σ''_v(:parent•**lista_T_c**) = σ_v(:parent•**lista_T_c**) ∪ { :new }.

Seja $\mathbf{t}_{novo}$ a tupla inserida em R_{ℓ_n} por *U*, e $\mathbf{t}_{pai}$ a tupla em R_b tal que $\mathbf{t}_{pai} \equiv$:parent.

Do procedimento **GVA**$_1$, temos que:

(3) $\mathbf{t}_{novo} \bullet \varphi^{-1} = \mathbf{t}_{pai}$, i.é., $\mathbf{t}_{novo} \in \mathbf{t}_{pai} \bullet \varphi$. Segue do caso 1 do procedimento **GVA**$_1$., como mostrado abaixo:

- **Caso 1.1**: Nesse caso temos que $\mathbf{t}_{novo} \bullet \mathbf{g}_q^{\ell_1} =$:parent$\bullet \mathbf{d}_q$, $1 \bullet q \leq m_1$ .Da ACO $[\mathbf{T}_{R_b}, \{\mathbf{f}_1^{\ell_1}, ..., \mathbf{f}_{m_1}^{\ell_1}\}] \equiv [\mathbf{T}_v, \{\mathbf{d}_1,..,\mathbf{d}_{m_1}\}]$, temos que $\mathbf{t}_{pai} \bullet \mathbf{f}_q^{\ell_1} =$:parent$\bullet \mathbf{d}_q$, $1 \bullet q \leq m_1$,. Então temos, $\mathbf{t}_{novo} \bullet \mathbf{g}_q^{\ell_1} = \mathbf{t}_{pai} \bullet \mathbf{f}_q^{\ell_1}$, $1 \bullet q \leq m_1$. Dado que ℓ_1 é inversa da chave estrangeira dada por $\mathbf{R}_{\ell_1}[\mathbf{g}_1^{\ell_1}, ..., \mathbf{g}_{m_1}^{\ell_1}] \subset \mathbf{R}_b[\mathbf{f}_1^{\ell_1}, ..., \mathbf{f}_{m_1}^{\ell_1}]$, então temos que $\mathbf{t}_{novo} \bullet \ell_1^{-1} = \mathbf{t}_{pai}$. Logo, $\mathbf{t}_{novo} \in \mathbf{t}_{pai} \bullet \ell_1$.
- **Caso 1.2**: Seja $\mathbf{t}_{R_{\ell_{n-1}}}$ a tupla em $R_{\ell_{n-1}}$ tal que $\mathbf{t}_{R_{\ell_{n-1}}} = \mathbf{t}_{pai} \bullet \ell_1 \bullet \ ... \ \bullet \ell_{n-1}$. Do caso 1.2 de **GVA**$_1$, temos que $\mathbf{t}_{novo} \bullet \mathbf{g}_q^{\ell_n} = \mathbf{t}_{R_{\ell_{n-1}}} \bullet \mathbf{f}_q^{\ell_n}$, $1 \bullet q \leq m_n$,. Dado que ℓ_n é inversa da chave estrangeira dada por $\mathbf{R}_{\ell_{n-1}}[\mathbf{f}_1^{\ell_n}, ...,\mathbf{f}_{m_n}^{\ell_n}] \subset \mathbf{R}_{\ell_n}[\mathbf{g}_1^{\ell_n},..,\mathbf{g}_{m_n}^{\ell_n}]$, então temos que $\mathbf{t}_{novo} \bullet \ell_n^{-1} = \mathbf{t}_{R_{\ell_{n-1}}}$. Logo, $\mathbf{t}_{novo} \in \mathbf{t}_{R_{\ell_{n-1}}} \bullet \ell_n$. Como $\mathbf{t}_{R_{\ell_{n-1}}} = \mathbf{t}_{pai} \bullet \ell_1 \bullet ... \bullet \ell_{n-1}$ então temos que $\mathbf{t}_{novo} \in \mathbf{t}_{pai} \bullet \ell_1 \bullet \ ... \ \bullet \ell_{n-1} \bullet \ell_n$

(4) $\mathbf{t}_{novo} \equiv$:new, i.é, $\mathbf{t}_{novo}$ satisfaz todas as assertivas de correspondência de $\mathbf{T_c}$ & $\mathbf{T}_{R_{\ell_n}}$. Segue dos casos 2-6 do procedimento **GVA**$_1$, como mostrado abaixo:

- **Caso 2**: Do Caso 2 de **GVA**$_1$ temos que $\mathbf{t}_{novo} \bullet \mathbf{b} =$ **:new** $\bullet \mathbf{a}$. Logo a ACC $[\mathbf{T_c} \bullet \mathbf{a}] \equiv [\mathbf{T}_{R_{\ell_n}} \bullet \mathbf{b}]$, é satisfeita.
- Os Casos 3 e 4 são similares ao Caso 2.
- **Caso 5**: Seja $\mathbf{t}_{R_{\ell'_p}}$ a tupla em $R_{\ell'_p}$ tal que $\mathbf{t}_{R_{\ell'_p}} \bullet \mathbf{k} =$ **:new** $\bullet \mathbf{a}$. Seja $\mathbf{t}_{R_{\ell'_1}}$ a tupla em $R_{\ell'_1}$ tal que $\mathbf{t}_{R_{\ell'_1}} \bullet \ell'_2 ... \bullet \ell'_p = \mathbf{t}_{R_{\ell'_p}}$. Do Caso 5 de **GVA**$_1$, temos que $\mathbf{t}_{novo} \bullet \mathbf{f}_q^{\ell'_1} = \mathbf{t}_{R_{\ell'_1}} \bullet \mathbf{g}_q^{\ell'_1}$, $1 \bullet q \leq m_1$..Dado que ℓ'_1 é uma chave estrangeira dada por $R_{\ell_n}[\mathbf{f}_1^{\ell'_1}, ..., \mathbf{f}_{m_1}^{\ell'_1}] \subset R'_{\ell_1}[\mathbf{g}_1^{\ell'_1},.., \mathbf{g}_{m_1}^{\ell'_1}]$, então temos que $\mathbf{t}_{novo} \bullet \ell_1 = \mathbf{t}_{R_{\ell'_1}}$ Como $\mathbf{t}_{R_{\ell'_1}} \bullet \ell'_2 \bullet ... \bullet \ell'_p = \mathbf{t}_{R_{\ell_p}}$ então temos que $\mathbf{t}_{novo} \bullet \ell'_1 \bullet \ell'_2 \bullet .. \bullet \ell'_p = \mathbf{t}_{R_{\ell'_p}}$. Como $\mathbf{t}_{R_{\ell'_p}} \bullet \mathbf{k} =$ **:new** $\bullet \mathbf{a}$ então, $\mathbf{t}_{novo} \bullet \ell'_1 \bullet \ell'_2 ... \bullet \ell'_p \bullet \mathbf{k} =$ **:new** $\bullet \mathbf{a}$. Logo, temos que a ACC $[\mathbf{T_c} \bullet \mathbf{a}] \equiv [\mathbf{T}_{R_{\ell_n}} \bullet \varphi' \bullet \mathbf{k}]$ é satisfeita.
- **Caso 6.1.1**: Seja $\mathbf{t}_{R_{\ell'_1}}$ a tupla em $R_{\ell'_1}$ tal que $\mathbf{t}_{R_{\ell'_1}} \equiv$ **:new** $\bullet \mathbf{a}$. Da ACO $[\mathbf{T}_a, \{\mathbf{h}_1,..,\mathbf{h}_{m_1}\}] \equiv [\mathbf{T}_{R_{\ell'_1}}, \{\mathbf{g}_1^{\ell'_1},.., \mathbf{g}_{m_1}^{\ell'_1}\}]$ temos que $\mathbf{t}_{R_{\ell'_1}} \bullet \mathbf{g}_q^{\ell'_1} =$ **:new** $\bullet \mathbf{a} \bullet \mathbf{h}_q$,$1 \bullet q \leq m_1$. Do Caso 6.1.1 de

GVA$_1$, temos que $\mathbf{t_{novo}} \bullet \mathbf{f}_q^{\ell'_1} = \mathbf{:new} \bullet \mathbf{a} \bullet \mathbf{h}_q$,$1 \bullet q \leq m_1$. Logo temos que $\mathbf{t_{novo}} \bullet \mathbf{f}_q^{\ell'_1} = \mathbf{t}_{R_{\ell'_1}} \bullet \mathbf{g}_q^{\ell'_1}$,$1 \bullet q \leq m_1$. Dado que ℓ'_1 é uma chave estrangeira dada por $R_{\ell_n}[\mathbf{f}_1^{\ell'_1}, ..., \mathbf{f}_{m_1}^{\ell'_1}] \subset R'_{\ell_1}[\mathbf{g}_1^{\ell'_1},.., \mathbf{g}_{m_1}^{\ell'_1}]$, então temos que $\mathbf{t_{novo}} \bullet \ell_1 = \mathbf{t}_{R_{\ell'_1}}$. Como $\mathbf{t}_{R_{\ell'_1}} \equiv \mathbf{:new} \bullet \mathbf{a}$, então $\mathbf{t_{novo}} \bullet \ell'_1 \equiv \mathbf{:new} \bullet \mathbf{a}$. Logo, temos que a ACC $[\mathbf{T_c} \bullet \mathbf{a}] \equiv [\mathbf{T}_{R_{\ell n}} \bullet \varphi']$ é satisfeita.

- **Caso 6.1.2**: Seja $\mathbf{t}_{R_{\ell'_p}}$ a tupla em $R_{\ell'_p}$ tal que $\mathbf{t}_{R_{\ell'_p}} \equiv \mathbf{:new} \bullet \mathbf{a}$. Da ACO $[\mathbf{T_a}, \{\mathbf{h}_1,..,\mathbf{h}_m\}] \equiv [\mathbf{T}_{R_{\ell'_p}}, \{\mathbf{e}_1,..,\mathbf{e}_m\}]$ temos que $\mathbf{t}_{R'_{\ell'_p}} \bullet \mathbf{e_i} = \mathbf{:new} \bullet \mathbf{a} \bullet \mathbf{h_i}$, $1 \bullet i \leq m$. Seja $\mathbf{t}_{R_{\ell'_1}}$ a tupla em $R_{\ell'_1}$ tal que $\mathbf{t}_{R_{\ell'_1}} \bullet \ell'_2 ... \bullet \ell'_p = \mathbf{t}_{R_{\ell'_p}}$. Do Caso 6.1.2 de **GVA**$_1$, temos que $\mathbf{t_{novo}} \bullet \mathbf{f}_q^{\ell'_1} = \mathbf{t}_{R_{\ell'_1}} \bullet \mathbf{g}_q^{\ell'_1}$, $1 \bullet q \leq m_1$. Dado que ℓ'_1 é uma chave estrangeira dada por $R_{\ell_n}[\mathbf{f}_1^{\ell'_1}, ..., \mathbf{f}_{m_1}^{\ell'_1}] \subset R'_{\ell_1}[\mathbf{g}_1^{\ell'_1},.., \mathbf{g}_{m_1}^{\ell'_1}]$, então temos que $\mathbf{t_{novo}} \bullet \ell_1 = \mathbf{t}_{R_{\ell'_1}}$. Como $\mathbf{t}_{R_{\ell'_1}} \bullet \ell'_2 ... \bullet \ell'_p = \mathbf{t}_{R_{\ell_p}}$ então temos que $\mathbf{t_{novo}} \bullet \ell'_1 \bullet \ell'_2 ... \bullet \ell'_p = \mathbf{t}_{R_{\ell'_p}}$. Como $\mathbf{t}_{R_{\ell'_p}} \equiv \mathbf{:new} \bullet \mathbf{a}$ então, $\mathbf{t_{novo}} \bullet \ell'_1 \bullet \ell'_2 ... \bullet \ell'_p \equiv \mathbf{:new} \bullet \mathbf{a}$. Logo, temos que a ACC $[\mathbf{T_c} \bullet \mathbf{a}] \equiv [\mathbf{T}_{R_{\ell_n}} \bullet \varphi']$ é satisfeita.
- Os Casos 6.2.1 e 6.2.2 são similares aos Casos 6.1.1 e 6.1.2, respectivamente.

De U e (3) temos que:

(5) $\sigma'_s(\mathbf{t_{pai}} \bullet \varphi) = \sigma_s(\mathbf{t_{pai}} \bullet \varphi) \cup \{ \mathbf{t_{novo}} \}$.

Dado que $[\mathbf{T_v} \bullet \text{lista\_T}_c] \equiv [\mathbf{T}_{R_b} \bullet \varphi]$, e $:\text{parent} \equiv \mathbf{t}_{pai}$, temos que :

(6) $\sigma'_v(:\text{parent} \bullet \mathbf{lista\_T_c}) \bullet \sigma'_s(\mathbf{t_{pai}} \bullet \varphi)$.

De (4), (5) e (6) temos que:

(7) $\sigma'_v(:\text{parent} \bullet \mathbf{lista\_T_c}) = \sigma_v(:\text{parent} \bullet \mathbf{lista\_T_c}) \cup \{ :\text{new} \}$.

De (2) e (7) temos que:

$\sigma'_v(:\text{parent} \bullet \mathbf{lista\_T_c}) = \sigma''_v(:\text{parent} \bullet \mathbf{lista\_T_c})$,

como queríamos demonstrar.

A.1.2: Caso 2- Ligação multivalorada não é a última

A seguir mostramos que o tradutor AdiçãoEmLista_T_c_Caso2 da figura 5.14 realiza a atualização μ solicitada na visão; i.é, a adição do objeto :new do tipo **Tc** na coleção aninhada lista_Tc do objeto :parent da visão V de tipo **Tv** .

- Temos que provar que:

(1) $\sigma'_v(:parent \bullet \mathbf{lista\_T_c}) = \sigma''_v(:parent \bullet \mathbf{lista\_T_c})$. (vide Figura A.1).

De μ temos que:

(2) $\sigma''_v(:parent \bullet \mathbf{lista\_T_c}) = \sigma_v(:parent \bullet \mathbf{lista\_T_c}) \cup \{ :new \}$.

Seja $\mathbf{t}_{novo}$ a tupla inserida em R_{ℓ_j} por *U*, $\mathbf{t}_{pai}$ a tupla em R_b tal que $\mathbf{t}_{pai} \equiv$:parent., e $\mathbf{t}_{R_{\ell_n}}$ a tupla em R_{ℓ_n} tal que $\mathbf{t}_{R_{\ell_n}} \equiv$ **:new**.

Do procedimento **GVA**$_2$, temos que:

(3) $\mathbf{t}_{novo} \in \mathbf{t}_{pai} \bullet \varphi'$. Segue do caso 1 do procedimento **GVA**$_2$. A prova é similar a do caso 1 do procedimento **GVA**$_1$.

(4) $\mathbf{t}_{novo} \bullet \varphi'' = \mathbf{t}_{R_{\ell_n}}$. Segue do caso 2 do procedimento **GVA**$_2$. A prova é similar a prova do caso 6 do procedimento **GVA**$_1$.

De (3) e (4) temos que:

(5) $\mathbf{t}_{R_{\ell_n}} \in \mathbf{t}_{pai} \bullet \varphi$.

De U e (5) temos que:

(6) $\sigma'_s(\mathbf{t}_{pai} \bullet \varphi) = \sigma_s(\mathbf{t}_{pai} \bullet \varphi) \cup \{ \mathbf{t}_{R_{\ell_n}} \}$.

Dado que $[\mathbf{T_v} \bullet lista\_T_c \mathbf{]} \equiv [\mathbf{T}_{R_b} \bullet \varphi]$, e $:parent \equiv \mathbf{t}_{pai}$, temos que :

(7) $\sigma'_v(:parent \bullet \mathbf{lista\_T_c}) \bullet \sigma'_s(\mathbf{t}_{pai} \bullet \varphi)$.

Dado que $\mathbf{t}_{R_{\ell_n}} \equiv$ **:new**, de (6) e (7) temos que :

(8) $\sigma'_v(:parent \bullet \mathbf{lista\_T_c}) = \sigma_v(:parent \bullet \mathbf{lista\_T_c}) \cup \{ :new \}$.

De (2) e (8) temos que:

$\sigma'_v(:parent \bullet \mathbf{lista\_T_c}) = \sigma''_v(:parent \bullet \mathbf{lista\_T_c})$,

como queríamos demonstrar.

A.2: Tradutores para Remoção em Coleção Aninhada

A.2.1: Caso 1- Ligação multivalorada é a última

A seguir mostramos que o tradutor "RemoçãoEm lista_T_c_Caso1" da figura 5.19 realiza a atualização μ solicitada na visão; i.é, a remoção do objeto :old do tipo $\mathbf{T_c}$ na coleção aninhada lista_T_c do objeto :parent da visão **V** de tipo $\mathbf{T_v}$.

- Temos que provar que:

(1) $\sigma'_v(:parent \bullet \mathbf{lista\_T_c}) = \sigma''_v(:parent \bullet \mathbf{lista\_T_c})$ (vide Figura A.1).

De μ temos que

(2) $\sigma''_v(:parent \bullet \mathbf{lista\_T_c}) = \sigma_v(:parent \bullet \mathbf{lista\_T_c}) - \{ :old \}$.

Seja $\mathbf{t}_{R_{\ell n}}$ a tupla removida em $R_{\ell n}$ por U, e $\mathbf{t}_{pai}$ a tupla em R_b tal que $\mathbf{t}_{pai} \equiv$:parent.

Seja $\varphi = \ell_1 \bullet .. \bullet \ell_n$ .Dado que $\mathbf{t}_{R_b} \bullet \ell_1 \bullet .. \bullet \ell_n = \mathbf{t}_{R_{\ell n}}$ então $\mathbf{t}_{R_{\ell n}} \in \mathbf{t}_{pai} \bullet \varphi$.

De U temos que:

(3) $\sigma'_s(\mathbf{t_{pai}} \bullet \varphi) = \sigma_s(\mathbf{t_{pai}} \bullet \varphi) - \{\mathbf{t}_{R_{\ell n}}\}$.

Dado que $[\mathbf{T_v} \bullet lista\_T_c \,] \equiv [\, \mathbf{T}_{R_b} \bullet \varphi]$, e :parent$\equiv \mathbf{t}_{pai}$, temos que :

(4) $\sigma'_v (:parent \bullet lista\_T_c) \bullet \sigma'_s(\mathbf{t_{pai}} \bullet \varphi)$.

Da ACO de $\mathbf{T_c}$ & $\mathbf{T}_{R_{\ell n}}$ $[T_c, \{\mathbf{f}_1, \mathbf{f}_2, ..., \mathbf{f}_w\}] \equiv [\mathbf{T}_{R_{\ell n}} \{\mathbf{g}_1, \mathbf{g}_2, ..., \mathbf{g}_w\}]$, e dado que $\mathbf{t}_{R_{\ell n}} \bullet \mathbf{f}_k =$:old$\bullet \mathbf{g}_k$, $1 \leq k \leq w$, temos que:

(5) :old $\equiv \mathbf{t}_{R_{\ell n}}$

De (3),(4) e (5) temos que :

(6) $\sigma'_v(:parent \bullet \mathbf{lista\_T_c}) = \sigma_v(:parent \bullet \mathbf{lista\_T_c}) - \{ :old \}$.

De (2) e (6) temos que:

$\sigma'_v(:parent \bullet lista\_T_c) = \sigma''_v(:parent \bullet lista\_T_c)$,

como queríamos demonstrar.

A.2.2: Caso 2 - Ligação multivalorada não é a última

A seguir mostramos que o tradutor "RemoçãoEm lista_T_c_Caso2" da figura 5.21 realiza a atualização μ solicitada na visão; i.é, a remoção do objeto :old do tipo $\mathbf{T_c}$ na coleção aninhada lista_T_c do objeto :parent da visão V de tipo $\mathbf{T_v}$.

- Temos que provar que:

(1) σ_v(:parent•**lista_T_c**) = σ''_v(:parent•**lista_T_c**) (vide Figura A.1).

De μ temos que

(2) σ_v(:parent•**lista_T_c**)= σ''_v(:parent•**lista_T_c**) - { :old }.

Seja:

$\mathbf{t}_{R_{\ell n}}$ a tupla em $R_{\ell n}$ onde $\mathbf{t}_{R_{\ell n}} \equiv$:old, e

$\mathbf{t}_{pai}$ a tupla em R_b tal que $\mathbf{t}_{pai} \equiv$:parent.

Seja $\mathbf{t}_{R_{\ell j}}$ a tupla em $R_{\ell j}$ tal que $\sigma_s(\mathbf{t}_{pai} \bullet \ell_1 \bullet .. \bullet \ell_j) = \mathbf{t}_{R_{\ell j}}$ e $\sigma_s(\mathbf{t}_{R_{\ell j}} \bullet \ell_{j+1} \bullet .. \bullet \ell_n) = \mathbf{t}_{R_{\ell n}}$

Logo temos que :

(3) $\mathbf{t}_{R_{\ell j}} \in \sigma_s(\mathbf{t_{pai}} \bullet \varphi')$.

(4) $\sigma_s(\mathbf{t}_{R_{\ell j}} \bullet \varphi'') = \mathbf{t}_{R_{\ell n}}$.

De (3) e (4) temos que:

(5) $\mathbf{t}_{R_{\ell n}} \in \sigma_s(\mathbf{t}_{pai} \bullet \varphi)$.

De U e (5) temos que:

(6) $\sigma'_s(\mathbf{t_{pai}} \bullet \varphi) = \sigma_s(\mathbf{t_{pai}} \bullet \varphi) - \{ \mathbf{t}_{R_{\ell n}} \}$.

Dado que [$\mathbf{T_v}$•lista_T_c **] ≡ [** $\mathbf{T}_{R_b}$•φ] e :parent≡$\mathbf{t}_{pai}$, temos que :

(7) σ'_v (:parent•lista_T_c) • $\sigma'_s(\mathbf{t_{pai}} \bullet \varphi)$.

Da ACO de $\mathbf{T_c}$&$\mathbf{T}_{R_{\ell n}}$ [$\mathbf{T_c}$,{$\mathbf{f}_1$,$\mathbf{f}_2$,..,$\mathbf{f}_w$}] ≡ [$\mathbf{T}_{R_{\ell n}}$ {$\mathbf{g}_1$,$\mathbf{g}_2$,..,$\mathbf{g}_w$}] e dado que $\mathbf{t}_{R_{\ell n}} \bullet \mathbf{f}_k$=:old•$\mathbf{g}_k$, 1≤k≤w, então temos:

(8) :old **≡** $\mathbf{t}_{R_{\ell n}}$

De (6), (7) e (8) temos que:

(9) σ_v(:parent•**lista_T_c**) = σ'_v (:parent•**lista_T_c**) ∪ { :old }.

De (2) e (9) temos que:

σ_v(:parent•lista_T_c) = σ''_v(:parent•lista_T_c),

como queríamos demonstrar.

A.3: Inserção na Visão V

A seguir mostramos que o tradutor "AdicãoNaVisaoV" da figura 5.24 realiza a atualização μ solicitada na visão; i.é, a adição do objeto :new do tipo $\mathbf{T_v}$ na visão **V**.

- Temos que provar que:

(1) $\sigma'_v = \sigma''_v$ (vide Figura A.1).

De μ temos que:

(2) $\sigma''_v = \sigma_v \cup \{ :new \}$.

Seja $\mathbf{t}_{novo}$ a tupla inserida em R_b por *U*.

Do passo 1 e do procedimento $\mathbf{GVA_3}$, temos que $\mathbf{t}_{novo}$ satisfaz todas as assertivas de correspondência de $\mathbf{T_v}$ & $\mathbf{T}_{Rb}$, para os atributos de cujos valores foram atribuídos no Passo 1. (No Passo1 são atribuídos valores somente aos atributos de :new que são monovalorados ou multivaorados, cuja AC é da forma $[\mathbf{T_v \bullet a}] \equiv [\mathbf{T}_{R_b}\{\mathbf{b_1, b_2, ..., b_n}\}]$) . Segue dos casos 1-5 do procedimento $\mathbf{GVA_3}$. Essa prova é semelhante a dos casos 2-6 do procedimento $\mathbf{GVA_1}$ (Caso 1 de inserção em coleção aninhada).

Do passo 2 temos que $\mathbf{t}_{novo}$ satisfaz todas as assertivas de correspondência de $\mathbf{T_v}$ & $\mathbf{T}_{Rb}$, para os atributos multivalorados **a** com AC da forma $[\mathbf{T_v} \bullet$lista_T_c $\mathbf{]\equiv[}\ \mathbf{T}_{R_b} \bullet \varphi]$. De acordo com o passo 2 de U, para cada atributo multivalorado **Lista-T_c** do objeto :new é solicitado a adição dos objetos na coleção aninhada **Lista-T_c** do objeto inserido :new, isto é, do objeto da visão **V**. Note que o pedido de atualização na coleção aninhada é traduzido em atualizações no banco de dados usando o tradutor para adição na tabela aninhada **Lista-T_c** (AdiçãoEmLista_T_c) da visão V, o qual é gerado como discutido na Seção 5.3. Assim temos que **:new**•lista_$T_c \equiv \mathbf{t}_{novo} \bullet \varphi$.

Dado que $\mathbf{t}_{novo}$ satisfaz todas as assertivas de correspondência de $\mathbf{T_v}$ & $\mathbf{T}_{Rb}$, então:

(3) $\mathbf{t}_{novo} \equiv$:new

De U, temos que:

(4) $\sigma'_s = \sigma_s \cup \{ \mathbf{t_{novo}} \}$.

Da ACE [**V**] ≡ [$\mathbf{R_b}$] e de (4) temos que

(5) $\sigma'_v = \sigma_v \cup \{:new\}$

De (2) e (5) temos que:

$\sigma'_v = \sigma''_v$,

como queríamos demonstrar.

A.4 : Remoção na Visão V

A seguir mostramos que o tradutor "RemoçãoNaVisaoV" da figura 5.27 realiza a atualização μ solicitada na visão; i.é, a remoção do objeto :old do tipo $\mathbf{T_v}$ da visão V .

- Temos que provar que:

(1) $\sigma'_v = \sigma''_v$ (vide Figura A.1).

De μ temos que

(2) $\sigma''_v = \sigma_v - \{ :old \}$.

Seja $\mathbf{t}_{Rb}$ a tupla removida em R_b por *U*.

De U temos que:

(3) $\sigma'_s = \sigma_s - \{ \mathbf{t}_{Rb} \}$.

Da ACO de $\mathbf{T}_v$ &$\mathbf{T}_{Rb}$ [$\mathbf{T}_v$,{$\mathbf{c}_1$,$\mathbf{c}_2$,...,$\mathbf{c}_w$} ≡ $\mathbf{T}_{Rb}$ {$\mathbf{d}_1$,$\mathbf{d}_2$,...,$\mathbf{d}_w$}], e dado que $\mathbf{t}_{Rb} \bullet \mathbf{d}_k$ =:old•$\mathbf{c}_k$, 1≤ k ≤w, temos que:

(4) :old ≡ $\mathbf{t}_{Rb}$

Da ACE [**V**] ≡ [$\mathbf{R_b}$] , de (3) e (4) temos que:

(5) $\sigma'_v = \sigma_v - \{ :old \}$.

De (2) e (5) temos que:

$\sigma_v = \sigma''_v$,

como queríamos demonstrar.

A.5: Modificação de Atributos Monovalorados da Visão V

A.5.1: Caso 1 - ACC é do tipo $[T_v \bullet a] \equiv [T_{Rb} \bullet b]$

A seguir mostramos que o tradutor "ModificaVisaoVCaso1" da figura 5.29 realiza a atualização μ solicitada na visão; i.é, a modificação do atributo monovalorado **a** de um objeto :new do tipo T_v da visão V.

Seja t_v a tupla em σ'_v tal que $t_v \equiv$:new

Temos que provar que:

(1) $\sigma'_v (t_v \bullet a) = \sigma''_v (t_v \bullet a)$ (vide Figura A.1).

De μ temos que

(2) $\sigma''_v (t_v \bullet a) =$ **:new•a**

Seja t_{Rb} a tupla em R_b onde $t_{Rb} \equiv$:new.

De U e da ACO de T_v & T_{Rb} $[T_v, \{c_1, c_2, ..., c_w\} \equiv T_{Rb} \{d_1, d_2, ..., d_w]$, temos que:

(3) $\sigma'_s (t_{Rb} \bullet b) =$:new•a

Da ACC $[T_v \bullet a] \equiv [T_{Rb} \bullet b]$ e dado que $t_v \equiv$:new:

(4) $\sigma'_v (t_v \bullet a) = \sigma'_s (t_{Rb} \bullet b)$

De (3) e (4) temos que:

(5) $\sigma'_v (t_v \bullet a) =$:new•a

De (2) e (5) temos que:

$\sigma_v = \sigma''_v$,

como queríamos demonstrar.

A.5.2: Caso 2 - ACC é do tipo $[T_v \bullet a, \{a_1, a_2, ..., a_n\}] \equiv [T_{Rb}, \{b_1, b_2, ..., b_n\}]$

A seguir mostramos que o tradutor "ModificaVisaoVCaso2" da figura 5.30 realiza a atualização μ solicitada na visão; i.é, a modificação do atributo monovalorado **a** de um objeto :new do tipo $\mathbf{T_v}$ da visão V.

Seja $\mathbf{t_v}$ a tupla em σ'_v tal que $\mathbf{t_v} \equiv$:new

Temos que provar que:

(1) $\sigma'_v (\mathbf{t_v \bullet a \bullet a_i}) = \sigma''_v (\mathbf{t_v \bullet a \bullet a_i})$, $1 \leq i \leq n$; (vide Figura A.1)

De μ temos que

(2) $\sigma''_v (\mathbf{t_v \bullet a \bullet a_i}) =$:new$\bullet$**a**$\bullet$ $\mathbf{a_i}$, $1 \leq i \leq n$;

Seja $\mathbf{t_{Rb}}$ a tupla em $\mathbf{R_b}$ onde $\mathbf{t_{Rb}} \equiv$:new

De U e da ACO de $\mathbf{T_v}$ & $\mathbf{T_{Rb}}$ $[\mathbf{T_v}, \{\mathbf{c_1}, \mathbf{c_2}, ..., \mathbf{c_w}\} \equiv \mathbf{T_{Rb}}$ $\{\mathbf{d_1}, \mathbf{d_2}, ..., \mathbf{d_w}]$, temos que:

(3) $\sigma'_s (\mathbf{t_{Rb} \bullet b_i}) =$:new$\bullet$**a**$\bullet$ $\mathbf{a_i}$, $1 \leq i \leq n$;

Da ACC $[\mathbf{T_v \bullet a}, \{\mathbf{a_1}, \mathbf{a_2}, ..., \mathbf{a_n}\}] \equiv [\mathbf{T_{R_b}}$ $\{\mathbf{b_1}, \mathbf{b_2}, ..., \mathbf{b_n}\}]$ e dado que $\mathbf{t_v} \equiv$:new**:**

(4) $\sigma'_v (\mathbf{t_v \bullet a \bullet a_i}) = \sigma'_s (\mathbf{t_{Rb} \bullet b_i})$, $1 \leq i \leq n$;

De (3) e (4) temos que:

(5) $\sigma'_v (\mathbf{t_v \bullet a \bullet a_i}) =$:new$\bullet$**a**$\bullet$ $\mathbf{a_i}$, $1 \leq i \leq n$;

De (2) e (5) temos que:

$\sigma_v = \sigma''_v$,

como queríamos demonstrar.

A.5.3: Caso 3 - ACC é do tipo $\mathbf{T_v \bullet a \equiv T_{Rb} \bullet \varphi \bullet k}$

A seguir mostramos que o tradutor "ModificaVisaoVCaso3" da figura 5.33 realiza a atualização μ solicitada na visão; i.é, a modificação do atributo monovalorado **a** de um objeto :new do tipo $\mathbf{T_v}$ da visão V.

Seja $\mathbf{t_v}$ a tupla em σ'_v tal que $\mathbf{t_v} \equiv$:new

Temos que provar que:

(1) $\sigma'_v (\mathbf{t_v \bullet a}) = \sigma''_v (\mathbf{t_v \bullet a})$ (vide Figura A.1).

De μ temos que

(2) σ''_v **(t_v•a)** = **:new•a**

Seja $\mathbf{t}_{Rb}$ a tupla em $\mathbf{R}_b$ onde $\mathbf{t}_{Rb} \equiv$:new

Seja:

$\mathbf{t}_{Rb}$ a tupla em $\mathbf{R}_b$ onde $\mathbf{t}_{Rb} \equiv$:new,

$\mathbf{t}_{R_{\ell_j}}$ a tupla em R_{ℓ_j} onde $\mathbf{t}_{R_{\ell_j}} = \mathbf{t}_{Rb} \bullet \ell_1 \bullet .. \bullet \ell_j$.

De U temos que:

(3) $\sigma'_s(\mathbf{t}_{R_{\ell_j}} \bullet \ell_{j+1} \bullet .. \bullet \ell_n \bullet \mathbf{K}) =$ **:new•a,**

Da ACC $[\mathbf{T_v \bullet a}] \equiv [\mathbf{T}_{Rb} \bullet \varphi \bullet \mathbf{K}]$, e de (3) temos que:

(4) σ'_v **(t_v•a) = :new•a**

De (2) e (4) temos que:

σ'_v **(t_v•a)** = σ''_v **(t_v•a)**

como queríamos demonstrar.

Referências Bibliográficas

[1] World Wide Web Consortium, "Extensible Markup Language (XML) Version 1.1", W3C Recommendation, 04th February 2004. Disponível em http://www.w3.org/ TR/2004/REC-xml11-20040204. Acesso em: 25 de março de 2004.

[2] DEUTSCH, A.; FERNANDEZ, M.; FLORESCU, D.; LEVY, A.; SUCIU, D. *XML-QL: A Query Language for XML*. 8 th International WWW Conference, Toronto, May 1999.

[3] World Wide Web Consortium. *XML Path Language (XPath) Version 1.0*. W3C Recommendation, 16 November 1999. Disponível em: <http://www.w3.org/TR/xpath>. Acesso em: 21 abril 2004.

[4] World Wide Web Consortium. *XQuery 1.0: An XML Query Language*. W3C Working Draft, 23 July 2004. Disponível em: <http://www.w3.org/TR/xquery/>. Acesso em: 8 abril 2004.

[5] CAREY, M., et al. XPERANTO: Publishing object-relational data as XML. In: *Proceedings Third International Workshop on the Web and Databases*. Dallas, Texas: May 2000, pages 105-110.

[6] SHANMUGASUNDARAM, J., et al. Querying XML Views of Relational Data. In: *Proceeding of 27th VLDB Conference*. Roma, Italy: 2001, pages 261-270.

[7] SHANMUGASUNDARAM, J., et al. XPERANTO: Bridging Relational Tecnology and XML. *IBM Research Report*. June, 2001.

[8] FERNÁNDEZ, M.; KADIYSKA, Y.; SUCIU, D.; MORISHIMA, A.; TAN, W.. SilkRoute: A Framework for Publishing Relational Data in XML. In: *ACM Transactions on Database Systems*, Vol. 27, Nº. 4, December 2002, Pages 438-493.

[9] FERNÁNDEZ, M.; TAN, W.; SUCIU, D.. Silkroute: Trading between relations and XML. In: *Proceedings of the Ninth International World Wide Web Conference,* Amsterdam: May 2000.

[10] ABITEBOUL, S.; BUNEMAN, P.; SUCIU, D.. *Gerenciando Dados na WEB*. Rio de Janeiro: Campos, 2000.

[11] BARU, C.; GUPTA, A.; LUDASCHER, B.; MARCIANO, R.; PAPAKOSTATINOU, Y.; VELIKHOV, P.; CHU, V.. XML-Based Information Mediation with MIX. In: *Proceddings of ACM SIGMOD Conf. on Management of Data*, p. 597-599, Philadelphia, Pennsylvania, USA, June 1999.

[12] MELLO, R. S.; HEUSER, C. A. A Bottom-Up Approach for Integration of XML Sources. In: *WIIW International Workshop on Data Integration on the Web*. Rio de Janeiro, Brazil, 2001.

[13] VIDAL, V. M. P.; LÓSCIO, B. F., SALGADO, A. C. Using correspondence assertions for specifying the semantics of xml-based mediators. In: WIIW 2001 -International Workshop on Information Integration on the Web – Technologies and Applications, Rio de Janeiro, 2001. *Proceedings of the International Workshop on Information Integration on the Web*. Rio de Janeiro: 2001, pages 3 – 11.

[14] LÓSCIO, B. F.. *Atualização de múltiplas bases de dados através de mediadores*. Fortaleza. Dissertação de Mestrado em Ciência da Computação, Universidade Federal do Ceará - UFC, 1998.

[15] Oracle Object Views. Oracle Corporation, 2001. Disponível em: http:// technet.oracle.com/ doc/server.815/a67781/c13obvw.htm

[16] TATARINOVI, I.; IVES, Z.G.; HALEVY, A .Y.; Weld, D.S. "Updating XML". In Proceedings of Sigmod, May de 2001.

[17] BANCILHON, F; SPYRATOS, N., "Updates semantics and relational view," *ACM Transactions on Database Systems*, vol. 6, December 1993.

[18] DAYAL, U.; BERNSTEIN A., "On the correct translation of update operations on relational views," *ACM Transactions on Database Systems*, vol. 7, Setembro 1982.

[19] FURTADO, A. L; CASANOVA, M. A, "Updating relational views query processing in database system," *New York: Ed. Springer Verlag*, 1985.

[20] MEDEIROS, C; TOMPA, F.W., "Understanding the implications of view update policies," *In Proc. of 11th International Conference on Very Large Databases*, 1985.

[21] KELLER, A. M.; "The role of semantics in translating view updates," *IEEE Computer*, vol.19, p. 63–73, January 1986.

[22] BARSALOU,T.; KELLER, A. M.; et al., "Updating relational databases through object-based views," SIGMOD Conference, p. 248–257, 1991.

[23] KELLER, A. M.; WIEDERHOLD, G., "Penguim: Objects for programs, relations for persistence," p. 75–88, Julho 2000.

[24] WIEDERHOLD, G.; BARSALOU, T.; LEE, B. S.; SIAMBELA, N.; SUJANSKY, W. “Use of relational storage and a semantic model to generate objects: The penguin project,” Database’91: Merging Policy, Standards and Technology, p. 503–515, 1991.

[25] BRAGANHOLO, V.; DAVIDSON, S.; HEUSER, C. UXQuery: building updatable XML views over relational databases. In: Brazilian Symposium on Databases, Manaus, AM, Brazil, Proceedings… 2003. p. 26-40.

[26] BRAGANHOLO, V.; DAVIDSON, S.; HEUSER, C. From XML View Updates to Relational View Updates: old solutions to a new problem. In: International Conference on Very Large Data Bases, Toronto, Canada, 2004.

[27] BRAGANHOLO, V.; DAVIDSON, S.; HEUSER, C. On the updatability of XML views over relational databases. In: WebDB, held in conjunction with SIGMOD/PODS, San Diego, California, 2003. Informal Proceedings… p. 31-36.

[28] TATARINOVI, I.; IVES, Z.G.; HALEVY, A .Y.; Weld, D.S. "Updating XML". In Proceedings of SIGMOD 2001, May de 2001.

[29] RYS, M. Bringing the Internet to Your Database: Using SQL Server 2000 e XML to Build Loosely-Coupled Systems. In: *Proceedings of the International Conference on Data Engineering*. Heidelberg, Germany: 2001, p. 465-472.

[30] HAYASHI, L. S.; HATTON, J. Combining UML, XML and Relational Database Tecnologies – The Best of All Worlds for Robust Linguistic Databases.In: *Proceedings of the IRCS Workshop on Linguistic Databases*. Philadelphia, USA: University of Pennsylvania, December 2001, p. 115-124.

[31] MUENCH, S. *Building Oracle XML Applications*. O’Reilly, September 2000.

[32] UMESHWAR, D.; BERNSTEIN, P. A. On the correct translation of update operations on relational views. *ACM Transactions on Database Systems*, 8(2):381-416, September 1982.

[33] Oracle Corporation. Disponível em: <http://technet.oracle.com>. Acesso em: 19 maio 2004.

[34] World Wide Web Consortium. *XSL Transformation (XSLT) Version 1.0*. W3C Recommendation, 16 November 1999. Disponível em: <http://www.w3.org/TR/xslt>. Acesso em: 22 Setembro 2003.

[35] SANTOS, L. A. L.. XML *Publisher*: Um *Framework* para Publicar Dados Objeto Relacionais Através de Visões XML. Fortaleza. Dissertação de Mestrado em Ciência da Computação, Universidade Federal do Ceará, Agosto/2004.

[36] O. Corporation, "PL/SQL - User's Guide and Reference", vol. Release 2 (9.2) - A96624-01, March, 2002. Disponível em: <http://downloadwest. oracle.com/docs/cd/B10501_01/appdev.920/a96624.pdf>. Acesso em: 12 Agosto 2004.

[37] COSTA, J. P. – Gerando Tradutores para Atualização de Banco de Dados através de Visões de Objetos. Dissertação de Mestrado, Universidade Federal do Ceará, 2002.

[38] World Wide Web Consortium. *XML Schema Part 1: Structures*. W3C Recomendation, 02 May 2001. Disponível em: <http://www.w3.org/TR/xmlschema-1/>. Acesso em: 15 Outubro 2003.

[39] World Wide Web Consortium. *XML Schema Part 2: Datatypes*. W3C Recomendation, 02 May 2001. Disponível em:

<http://www.w3.org/TR/xmlschema-2/>. Acesso em: 17 Agosto 2003.

[40] O. Corporation, *Oracle9i - Application Developer's Guide - Object-Relational Features*, vol. Release 2 (9.2) - A96594-01, March 2002. Disponível em: <http://download-west.oracle.com/docs/cd/B10501_01/appdev.920/a96594.pdf >.Acesso em: 27 March 2004.

[41] SILBERSCHARTZ, A.; KORTH, H. F. Sistemas de Banco de Dados. McGraw Hill, 3ª ed., 1998.

[42] FURLAM, J. D. Modelagem de Objetos Através de XML. Análise e Desenho Orientados a Objeto. Makron Books, 1998.

[43] ZEDULKA, J. Object-relational modeling in UML. In: *Proceedings of the Conference*. Information Systems Modelling, Ostrava, CZ, March, 2001, p. 17-24.

[44] ELMASRI, R.; NAVATHE, S. *Fundamentals of Database Systems*. Addison-Weslley, 3ª ed., 2000.

[45] *World Wide Web Consortium*. Disponível em: <http://www.w3.org>. Acesso em: 7 Janeiro 2004.

[46] World Wide Web Consortium. *HTML Version 4.0.1*. W3C Recommendation, 24 December 1999. Disponível em: <http://www.w3.org/TR/html4/>. Acesso em: 4 Dezembro 2003.

[47] SAHUGUET, A.. Everything you ever wanted to know about dtds, but were afraid to ask. International Workshop on the Web and Databases. In: *Proceedingsv Of WebDB'2000*, pages 69-74.

[48] VILAS BOAS, R. M. F. XMLS+Matcher: Um método para matching de XML Schemas Semânticos. Dissertação de Mestrado, Universidade Federal do Ceará, Agosto de 2002.

[49] World Wide Web Consortium. *Resource Description Framework* (RDF): Concepts and Abstract Syntax. W3C Recommendation, 10 February 2004. Disponível em: <http://www.w3.org/TR/rdf-concepts/>. Acesso em: 20 Março 2004.

[50] LEE, D.; CHU, W. Comparative Analysis of Six XML Schema Language. In: *ACM SIGMOD Record*, p.76-87, vol. 29, nº 3, September 2000.

[51] DUCKETT, J.; GRIFFIN, O.; MOHR, S.; NORTON, F.; STOKES-REES, I.; WILLIAMS, K.; CAGLE, K.; OZU, N.; TENNISON, J.. *Professional XML Schemas*. Wrox Press Ltd. Birmingham, 2001.

[52] FLORESCU, D.; DEUTSCH, A.; LEVY, A.; SUCIU, D.; FERNANDEZ, M.. A Query Language for XML. In: *Proceedings of Eighth International Conference on World Wide Web*. Toronto, Canadá, 1999, pages 1155-1169.

[53] ABITEBOUL,S. et al. "The Lorel query language for semistructured data", International Journal on Digital Libraries, vol. 1, no.1, p.66-68, 1997.

[54] VIDAL, V.M.P, LÓSCIO, B.F.: *Solving the Problem of Semantic Heterogeneity in Defining Mediator Update Translators*. In Proc. of 18th Intern. Conf. on Conceptual Modeling, Paris, France, p. 293-308, 1999.

[55] TAKAHASHI, T.; KELLER, A.M.. *Implementation of object view query on relational database. In: Int'l Conf. on Data and Knowledge Systens for Manufaturing and Engineering - DKSME*. Hong Kong, May 1994.

[56] MADHAVAN, J.; BERNESTEIN, P.; RAHM, E. Generic Schema Matching

with Cupid. In: *Proceddings of VLDB*. 2001, pages 49-58.

[57] POPA, L.; VELEGRAKIS, Y.; MILLER, R. J.; HERNANDEZ, M. A.; FAGIN, R. Translating Web Data. In: *VLDB*. August 2002, p. 598–609.

[58] RAHM, E.; BERNSTEIN, P. A. A Survey of Approaches to Automatic Schema Matching. In: *VLDB Journal*, 2001, p. 334–350.

[59] AHO, A. V.; SETHI, R.; ULLMAN, J.D. *Compiladores: Princípios, Técnicas e Ferramentas*. LTC, 1995.