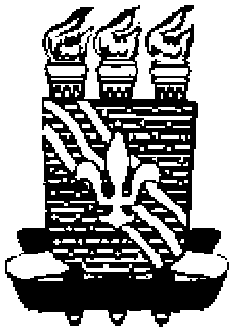

Universidade Federal da Paraíba - UFPB
Centro de Ciências e Tecnologia - CCT
Coordenação de Pós-Graduação em Informática - COPIN

Um Sistema de Controle de Integridade para um Modelo de Dados Aberto

Marinaldo Nunes da Silva

Campina Grande
Agosto de 2000

Marinaldo Nunes da Silva

Um Sistema de Controle de Integridade para um Modelo de Dados Aberto

Dissertação submetida ao Curso de Pós-Graduação em Informática do Centro de Ciências e Tecnologia da Universidade Federal da Paraíba, em cumprimento às exigências parciais para a obtenção do grau de Mestre em Informática.

Orientador: Ulrich Schiel
Linha de Pesquisa: Sistemas de Informação e Bancos de Dados
Área de Concentração: Ciência da Computação

Campina Grande
Agosto de 2000

FICHA CATALOGRÁFICA

S586S

SILVA, Marinaldo Nunes da

Um Sistema de Controle de Integridade para um Modelo de Dados Aberto.

Dissertação de Mestrado, Universidade Federal da Paraíba, Centro de Ciências e Tecnologia, Coordenação de Pós-Graduação em Informática, Campina Grande – PB, Agosto de 2000.

124 p. Il.

Orientador: Ulrich Schiel

Palavras Chaves:

1. Banco de Dados Ativos
2. Restrição de Integridade
3. Modelos de Dados Orientados a Objeto

CDU – 681.3.07B

UM SISTEMA DE CONTROLE DE INTEGRIDADE PARA UM MODELO DE DADOS ABERTO

MARINALDO NUNES DA SILVA

DISSERTAÇÃO APROVADA EM 28.08.2000

PROF. ULRICH SCHIEL , Dr.
Orientador

PROF. MARCUS COSTA SAMPAIO , Dr.
Examinador

PROF. DÉCIO FONSECA, Dr.
Examinador

CAMPINA GRANDE – PB

Aos meus pais, Luiza e Otávio, pelo trabalho e apoio irrestrito para que eu consiga atingir meus objetivos.

RESUMO

Este trabalho dedicou-se ao desenvolvimento de um sistema no topo de um SGBD relacional, o GeRATOM, para o controle de integridade de dados em um modelo de dados aberto (TOM), viabilizando o processamento de regras de integridade implícitas predefinidas do modelo e de novas regras que venham a ser incluídas no sistema à medida que o modelo é modificado.

O sistema realiza esta funcionalidade através da especificação de regras ECA, o que garante flexibilidade quanto ao número de restrições implícitas que podem ser definidas. Mas, diferentemente da maioria dos sistemas de regras existentes, que definem regras em relação a determinadas classes de uma aplicação, no GeRATOM, também é possível especificar regras genéricas associadas aos conceitos do modelo.

Com o GeRATOM, conseguimos acrescentar aos SGBDs um conjunto de conceitos não disponíveis em suas implementações, aumentando a expressividade de seus modelos de dados e diminuindo a diferença entre a definição conceitual das aplicações e suas implementações nos SGBDs.

AGRADECIMENTOS

A Deus, por ter me dado força e determinação para terminar este trabalho.

Ao meu orientador Ulrich Schiel, pela orientação e conhecimento transmitido durante a realização deste trabalho.

Aos colegas do mestrado pelo incentivo e bons momentos que passamos juntos, em especial, Gilene, Adriano, Juliano, Carlos Eduardo, André e Júlio.

E às pessoas que contribuíram direta ou indiretamente para a realização deste trabalho. Gostaria de agradecer em particular a Zeneide, Manuela, Vera e Aninha.

M. N. S.

SUMÁRIO

1 - INTRODUÇÃO 1
1.1 - MOTIVAÇÃO 1
1.2 - OBJETIVO DA PESQUISA 7
1.3 - ESTRUTURA DO TRABALHO 8
2 – CONTROLE DE INTEGRIDADE E BANCOS DE DADOS ATIVOS 9
2.1 - CONTROLE DE INTEGRIDADE 9
2.1.1 - TIPOS DE RESTRIÇÕES DE INTEGRIDADE 9
2.1.2 - ESPECIFICAÇÃO DE RESTRIÇÕES IMPLÍCITAS 10
2.1.3 - ESPECIFICAÇÃO DE RESTRIÇÕES EXPLÍCITAS 11
2.1.4 - MODELOS DE DADOS E SUAS RESTRIÇÕES DE INTEGRIDADE 12
2.1.5 – RESTRIÇÕES DE INTEGRIDADE NO PADRÃO SQL-92 15
2.2 - BANCOS DE DADOS ATIVOS 18
2.2.1 - EVENTO 18
2.2.2 - CONDIÇÃO 19
2.2.3 - AÇÃO 19
2.2.4 – VALORES DE TRANSIÇÃO 19
2.2.5 - SEMÂNTICA DE EXECUÇÃO DE REGRAS 20
3 - TEMPORAL OBJECT MODEL (TOM) 23
3.1 - ELEMENTOS DO TOM 23
3.1.1 - CLASSE 23
3.1.2 - RELACIONAMENTO 24
3.1.3 - ABSTRAÇÕES HIERÁRQUICAS 24
3.1.4 - ASPECTOS TEMPORAIS 27
3.1.5 - MÉTODO 27
3.1.6 – OBJETOS VERSIONADOS 28
3.2 - OPERAÇÕES ELEMENTARES 28
3.3 - INTEGRIDADE SEMÂNTICA NO TOM 29
3.3.1 - PREDICADOS ESTRUTURAIS 30
3.3.2 - AXIOMAS DINÂMICOS E OS EFEITOS COLATERAIS 32
3.4 - METANÍVEL DO TOM 37
4 - OBJETIVO E ESPECIFICAÇÃO DO GERENCIADOR DE REGRAS ATIVAS DO TOM 39
4.1 - OBJETIVO DO GeRATOM 39
4.2 - ESPECIFICAÇÃO DO GeRATOM 39
4.2.1 - CONCEITOS 40
4.2.2 - AXIOMAS DINÂMICOS E EFEITOS COLATERIAS 41
4.2.3 - PREDICADOS ESTRUTURAIS 44
4.2.4 - REGRAS ATIVAS 45

4.3 - SEMÂNTICA DE EXECUÇÃO DE REGRAS....55
4.3.1 - MODOS DE ACOPLAMENTO....55
4.3.2 - PRIORIDADE....57
4.3.3 - TERMINAÇÃO....57
4.3.4 - DETECÇÃO DE EVENTOS....58
4.3.5 - DISPARO DE REGRAS CONSIDERANDO O FATOR CLASSE....58
4.3.6 - PREDICADOS ESTRUTURAIS, PASSAGEM DE PARÂMETROS E A AVALIAÇÃO DA CONDIÇÃO....59
5 – PROJETO DO GeRATOM....75
5.1 – PROJETO TOP....75
5.1.1 – ARQUITETURA DO GOTA....76
5.1.2 – CONVERSÃO DO MODELO TOM PARA O MODELO RELACIONAL....77
5.1.3 - DICTOM - DICIONÁRIO DO MODELO TOM....82
5.1.4 - MANIPULADOR DE OBJETOS....85
5.2 - O SISTEMA GeRATOM....92
5.2.1 – ARQUITETURA DO GeRATOM....92
5.2.2 - INTERAÇÃO ENTRE OS MÓDULOS DO SISTEMA....93
5.3 – GERENCIADOR DE REGRAS....94
5.4 - MONITOR DE EVENTOS....105
5.5 - PROCESSADOR DE REGRAS....105
5.6 - GERENCIADOR DE OPERAÇÕES....106
5.7 - TERMINAÇÃO....108
6 – CONCLUSÕES E PROPOSTAS PARA TRABALHOS FUTUROS....112
6.1 – CONSIDERAÇÕES FINAIS....112
6.2 – PROPOSTAS PARA TRABALHOS FUTUROS....113
BIBLIOGRAFIA....114
APÊNDICE A – LINGUAGEM DE DEFINIÇÃO DE DADOS DO TOM....118
APÊNDICE B – DESCRIÇÃO DA BASE DE REGRAS....122

LISTA DE FIGURAS

Figura 1.1 Dimensão de Ponto de Vista. 5
Figura 1.2 Dimensão de Intenção/Extensão. 5
Figura 2.2 Um algoritmo de processamento de regras. 20
Figura 3.1 Níveis de Abstração no TOM 38
Figura 4.1 Manipulação de Conceitos do TOM 40
Figura 4.2 Manipulação de ADs e ECs 41
Figura 4.3. Mantém Salário 43
Figura 4.4 Manipulação de Predicados Estruturais 44
Figura 4.5 Classes básicas do GeRATOM 45
Figura 4.7 Manipulação de Eventos 48
Figura 4.8 Classe Condição 48
Figura 4.9 Classe Ação 49
Figura 4.10 Classe Gatilho 51
Figura 4.11 Classe Regra 53
Figura 4.12 Colaboração entre as classes básicas no processamento de um regra. 55
Figura 5.1 Arquitetura do GOTA 76
Figura 5.2 Manipulador de Objetos 85
Figura 5.3 Janela para criação de um novo objeto. 86
Figura 5.4 Janela com relacionamentos 86
Figura 5.5 Janela para excluir um objeto. 87
Figura 5.6 Janela para estabelecer um relacionamento. 88
Figura 5.7 Janela para remover um relacionamento 89
Figura 5.8 Janela para alterar um relacionamento. 90
Figura 5.9 Janela para inserir um elemento em um grupo. 91
Figura 5.10 Janela para excluir um elemento de um grupo 92
Figura 5.11 Arquitetura do GeRATOM 93
Figura 5.12 Janela inicial do Gerenciador de Regras 94
Figura 5.13 Janela para inserir uma regra 96
Figura 5.14 Janela para associar regra à classe 97
Figura 5.15 Janela para excluir uma regra 97
Figura 5.17 Janela para inserir um evento 98
Figura 5.18 Janela para modificar status de eventos 99
Figura 5.19 Janela para inserir um novo gatilho 100
Figura 5.20 Janela para modificar gatilho 101

Figura 5.21 Janela para inserir um novo predicado..... 102
Figura 5.22 Janela para inserir um novo conceito..... 103
Figura 5.23 Janela para associar regra e conceito. 104
Figura 5.24 Janela para associar AD/EC a um BD 105
Figura 5.25 Arquitetura interna do Processador de Regras..... 106

LISTA DE TABELAS

Tabela 3.1 Operações Elementares 28
Tabela 3.2 Predicados Primitivos 30
Tabela 3.3 Predicados da Generalização/Especialização 31
Tabela 3.4 Predicados do Agrupamento 31
Tabela 3.5 Predicados da Agregação 31
Tabela 3.6 Predicados da Herança 32
Tabela 3.7 Predicados do Tempo 31
Tabela 4.4 Modos de Acoplamento EC/CA no GeRATOM 57

1 - INTRODUÇÃO

1.1 - MOTIVAÇÃO

Bancos de dados (BD) armazenam dados de uma determinada parte do mundo real, denominada universo do discurso (UD). Cada UD possui regras que devem ser obedecidas e, como os dados, especificadas e mantidas no BD.

Bancos de dados são definidos utilizando modelos de dados compostos por um conjunto de conceitos utilizados para descrever a estrutura do BD, ou seja, seus dados, relacionamentos e suas restrições. Além dos aspectos estruturais, também são definidas as operações de manipulação dos dados permitidas no modelo.

Os primeiros modelos de dados foram concebidos para resolver problemas puramente computacionais relacionados à manutenção dos dados, deixando em segundo plano a descrição conceitual das aplicações.

Esta orientação acarreta alguns problemas:

a) Dificuldade de representar estruturas inerentes à aplicação no BD
b) Dificuldade de captar toda a semântica da aplicação.

Buscando reduzir estes problemas foram concebidos os modelos de dados semânticos [7, 16, 17, 25], com o objetivo de captar o máximo de semântica das aplicações, abstraindo aspectos computacionais.

Com os modelos semânticos, os problemas associados aos primeiros modelos, aparentemente, foram resolvidos. Mas, apesar do avanço constatado, os SGBDs por muitos anos não acompanharam a evolução, pois ainda eram implementados considerando os primeiros modelos de dados.

Um avanço significativo está sendo dado com a geração atual de SGBDs chamados de OO-puros ou OR (Objeto-Relacional) que permitem a definição de novas classes ou tipos do usuário, permitindo assim, a modelagem das aplicações o mais próximo da realidade. A definição de novas classes ou tipos, na maioria das vezes, abrange a especificação estrutural e comportamental dos dados.

Com relação ao comportamento dos dados, sua modelagem é feita através de um conjunto de operações definidas pelo usuário e específicas para aquele conjunto de

dados. Então, podemos afirmar que, com os SGBD's OO e OR, toda vez que uma nova abstração é definida, além da descrição estrutural de seus dados, o usuário deve definir proceduralmente o comportamento dos dados segundo a nova abstração.

Considerando os SGBDs disponíveis, o projeto conceitual da aplicação é feito de forma a captar toda a semântica da aplicação, mas, no momento de sua implementação, a descrição obtida precisa ser mapeada para uma representação suportada pelo SGBD que, na maioria das vezes, não absorve toda a semântica descrita na modelagem conceitual.

Esta representação é feita através das restrições implícitas suportadas pelo modelo de dados do SGBD. Restrições implícitas são restrições que podem ser descritas diretamente pelo modelo e podem ser inseridas no esquema de definição do BD.

A não absolvição de toda a semântica de uma aplicação é provocada pela simplicidade dos modelos de dados dos SGBDs, o que tem como conseqüência, a baixa expressividade de suas restrições implícitas.

Então, quanto mais rico for o modelo de dados do SGBD, maior será a expressividade de suas restrições implícitas, consequentemente mais semântica da aplicação poderá ser descrita pelo próprio modelo de dados e menor será a necessidade de utilização de formas complementares às restrições implícitas para garantir a semântica das aplicações.

Formas complementares às restrições implícitas são utilizadas para verificar restrições que não podem ser descritas diretamente pelo modelo de dados. Estes tipos de restrições são denominados de restrições explícitas.

Maneiras distintas são utilizadas para verificar restrições explícitas, dentre as quais podemos destacar: sua inclusão nos programas aplicativos ou utilização de um mecanismo de *triggers*.

A primeira forma de garantir as restrições explícitas é inseri-las diretamente nos programas aplicativos, o que permite controlar qualquer tipo de restrição e defini-las nas operações apropriadas. A flexibilidade que a programação oferece é comprometida pela necessidade, imposta aos programadores, de conhecer, implementar e garantir a verificação das restrições e pela dificuldade em modificar ou incluir uma nova regra.

Os problemas relacionados à verificação das restrições através da programação estão ligados ao fato de que, se diferentes programas necessitarem verificar a mesma

restrição faz-se necessário implementá-la mais de uma vez, o que pode levar a diferentes formas de verificação de uma mesma restrição.

Ainda considerando a programação, se uma restrição for modificada, eliminada ou uma nova restrição for incluída, alterações em todos os programas que verificam a restrição modificada ou excluída devem ser realizadas, o mesmo acontecendo no caso da inclusão de uma nova restrição.

Outra forma utilizada para o controle de restrições explícitas é a utilização de *triggers* (gatilhos) para monitorar situações críticas das aplicações e tomar providências para manter a consistência dos dados.

Triggers permitem a especificação das restrições de maneira independente dos programas das aplicações, o que elimina os problemas da inclusão direta das restrições nos programas. Outra vantagem é a possibilidade de utilização de um mesmo *trigger* para verificar a mesma restrição em diferentes programas, como também, a inclusão ou exclusão de uma restrição sem afetar os programas das aplicações. Muitos SGBDs suportam mecanismos de *triggers* para verificação de restrições de integridade explícitas [3, 6, 19, 20, 29].

Apesar das vantagens dos *triggers* para a descrição das restrições de uma aplicação, existem alguns problemas quanto a sua implementação nos SGBDs. Entre os problemas podemos destacar a falta de padronização dos mecanismos de *triggers,* ou seja, cada SGBD implementa seus *triggers* de forma particular. Outro problema que podemos citar é que alguns SGBDs possuem restrições para a definição de *trigger*s, por exemplo, limite quanto ao número de *triggers* que podem ser definidos sobre um objeto do BD, limitações quanto aos tipos de eventos monitorados pelo sistema e operações que podem ser executadas como ações, restrições na especificação de suas condições, entre outros. Uma descrição dos mecanismos de *triggers* disponibilizados por diferentes SGBDs pode ser encontrada em [6].

Existem outras formas para a verificação de restrições explícitas, por exemplo, asserções e pré e pós-condições.

Com as asserções é possível descrever um conjunto expressivo de restrições, mas poucos SGBDs disponibilizam este recurso devido a dificuldade de implementá-lo de forma eficiente.

Já com as pré e pós-condições conseguimos especificar restrições de forma localizada, pois as condições são inseridas dentro do próprio programa o que leva a

necessidade de repetir a mesma condição em diferentes programas que precisam verificá-las.

Pelo que descrevemos, restrições de integridade implícitas e explícitas são verificadas de formas distintas. As restrições implícitas são verificadas diretamente pelos SGBDs, enquanto as restrições explícitas podem ser verificadas através de: programação, *triggers*, asserções, pré e pós-condições.

A verificação direta das restrições implícitas pelo SGBDs é possibilitada em virtude do número fixo de conceitos suportados pelos modelos de dados, consequentemente, o conhecimento prévio de quais restrições devem ser mantidas. Esta estratégia garante eficiência do controle das restrições, mas dificulta a expansão dos modelos de dados suportados pelos SGBDs, não permitindo que o modelo seja alterado sem que uma reimplementação do sistema seja necessária.

Esta abordagem para implementação de SGBDs é satisfatória para os modelos de dados fechados, ou seja, modelos cujo número de conceitos suportados não variam, mas é inadequado para modelos de dados abertos, que são modelos que disponibilizam um mecanismo para a inclusão de novos conceitos ao modelo.

A denominação “modelos de dados abertos” foi extraída da proposta de padronização para a implementação de um sistema de informação computadorizado [2], segundo duas dimensões ortogonais: a dimensão de ponto-de-vista e a dimensão de intenção/extensão.

A dimensão de ponto-de-vista (Figura 1.1) engloba três níveis de esquema: externo, conceitual e interno, onde cada nível repete a descrição do sistema, mas com uma finalidade diferente e tem a ele associado um ou mais esquemas de descrição do sistema.

Os esquemas externos (visões) descrevem o BD segundo visões de diferentes grupos de usuários. O esquema conceitual descreve o sistema como um todo integrado, ou seja, é uma visão global BD, escondendo detalhes de implementação. O esquema interno descreve as estruturas físicas de armazenamento do BD.

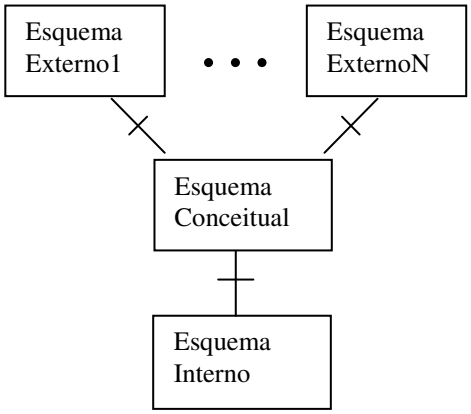

Figura 1.1 Dimensão de Ponto de Vista.

A dimensão de intenção/extensão possui quatro níveis de descrição (Figura 1.2) onde um nível intermediário corresponde à intenção do nível inferior e, ao mesmo tempo, a extensão do nível superior.

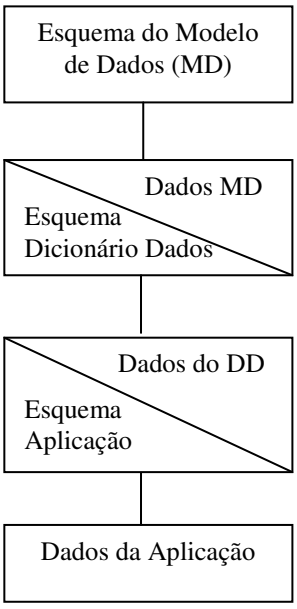

Figura 1.2 Dimensão de Intenção/Extensão.

Temos no primeiro nível (só extensional), os dados da aplicação. No segundo nível o esquema da aplicação (intencional) ou uma aplicação particular de um modelo de dados (extensional). No terceiro nível um modelo de dados e no quarto nível (só intencional) temos uma (meta-) linguagem para descrever modelos de dados, característica que determina que um modelo é aberto.

Note que a dimensão de intenção/extensão é ortogonal à anterior, ou seja, para cada esquema da dimensão de ponto-de-vista podemos ter os quatro níveis descritos.

Quanto ao tópico que nos interessa no presente trabalho, as regras de integridade, elas podem aparecer tanto no segundo nível quanto no terceiro nível.

As regras do segundo nível são chamadas de regras da aplicação e as do nível superior são as regras do modelo. Ambas são utilizadas, de fato, no primeiro nível, o nível da aplicação. Neste caso, as regras do modelo garantem que o usuário não faça operações inválidas ou incompletas em relação ao esquema conceitual descrito no modelo de dados.

Em um modelo aberto, os novos conceitos criados, da mesma forma que os conceitos dos modelos fechados, têm a eles associados, regras de integridade implícitas que precisam ser verificadas para manter a consistência dos dados que utilizam estes novos conceitos.

A verificação direta de suas restrições pelo sistema não é possível, uma vez que as restrições não são previamente conhecidas. Incluí-las dentro dos programas das aplicações não é desejável, pois cria a necessidade de implementar as regras toda vez que um conceito é utilizado. Então, como proceder para garantir a integridade dos dados nos modelos abertos?

A verificação de restrições implícitas feita diretamente pelos SGBDs não permite a modificação do modelo de dados sem a necessidade de modificar o sistema, o que não é aceitável para um modelo de dados aberto, que tem o objetivo de permitir que um administrador de modelos o modifique de acordo com a necessidade de suas aplicações.

Então, um sistema mais flexível para a realização do controle das restrições implícitas é recomendável.

O modelo de *triggers* é muito adequado para resolver este problema, pois oferece total flexibilidade para acrescentar ou remover regras de integridade do sistema, o que permite adequar o conjunto de restrições implícitas de acordo com as modificações impostas ao modelo.

Então, combinando a característica aberta de um modelo de dados com a flexibilidade de definição de regras oferecida pelos triggers, é possível estender um modelo de dados limitado semanticamente, ou seja, um modelo onde suas restrições implícitas não são capazes de retratar toda a realidade modelada, com novos conceitos

de forma a enriquecer seu conjunto de restrições implícitas e aproximá-lo da realidade que deve ser representada pelos dados de um BD.

Com esta estratégia é possível descrever, quase que por completo, um BD segundo as restrições implícitas do modelo, diminuindo assim, a necessidade de definição de restrições explícitas para garantir a semântica dos dados das aplicações. A expressão "quase que por completo" foi colocada uma vez que, por mais rico que um modelo de dados venha ser, não garantimos que todas as situações de uma realidade possam ser por ele modeladas.

1.2 - OBJETIVO DA PESQUISA

Na seção anterior descrevemos que durante o projeto de um BD sua definição conceitual é convertida para o modelo de dados suportado pelo SGBD e que, durante esta conversão, muito da semântica da aplicação é perdida, perdas estas, que são compensadas com a utilização de mecanismos para descrever restrições explícitas. A conversão de um modelo para outro ocorre em virtude das diferenças semânticas entre os modelos utilizados para descrever e implementar o BD.

Então, buscando diminuir a diferença entre os modelos de dados dos SGBDs e a realidade representada pelo BD, propomos neste trabalho, um sistema que poderá ser agregado a diferentes SGBDs de forma a enriquecer seus modelos de dados com novos conceitos tendo como conseqüência o aumento da expressividade de suas restrições implícitas, permitindo assim, que toda a descrição de um BD seja feita através deste tipo de restrição.

Considerando que atualmente a maioria dos SGBDs utilizados são relacionais, o sistema foi definido e implementado para enriquecer os modelos de dados de tais sistemas. Para isso, implementamos o sistema com base no padrão SQL/92 [8] que é suportado pela maioria dos SGBDs relacionais comerciais.

Ainda para a criação do sistema, utilizamos o modelo TOM [9, 24, 27], pois o mesmo oferece um mecanismo de metaclasses (metanível), caracterizando-o como um modelo de dados aberto.

Outra característica deste trabalho leva em consideração a necessidade de um sistema que possa reagir à mudanças de estados dos dados. Assim, implementamos um

sistema que tem a capacidade de reagir a possíveis modificações sobre os dados do BD. Sistemas com esta característica são denominados Bancos de Dados Ativos.

Alguns SGBDs disponibilizam funcionalidades ativas através de mecanismos de *triggers*, mas, devido a falta de padronização, a utilização de um destes mecanismos para implementar o sistema proposto criaria uma versão proprietário do sistema, ou seja, uma versão especifica para um único SGBD, o que comprometeria a portabilidade do sistema em relação a outros SGBDs.

1.3 - ESTRUTURA DO TRABALHO

O trabalho está estruturado da seguinte forma:

No capítulo 2, é feito um breve estudo sobre controle de integridade de dados e bancos de dados ativos. No capítulo 3, descrevemos o modelo TOM, destacando seus conceitos, o controle de integridade do modelo e seu metanível, mecanismo que o caracteriza como um modelo de dados aberto. No capítulo 4, descrevemos o objetivo e a especificação do Gerenciador de Regras Ativas do modelo TOM (GeRATOM). O capítulo 5 aborda aspectos de implementação do sistema destacando a implementação do DICTOM, Manipulador de Objetos e do próprio GeRATOM. E finalizando, no capítulo 6 temos as conclusões e as perspectivas para trabalhos futuros.

2 – CONTROLE DE INTEGRIDADE E BANCOS DE DADOS ATIVOS

2.1 - CONTROLE DE INTEGRIDADE

Um importante aspecto que deve ser considerado quando implementamos um banco de dados é a consistência de seus dados. A consistência dos dados é mantida através de um conjunto de regras, denominadas de restrições de integridade, que determinam condições que devem ser obedecidas pelos dados antes e depois que alguma modificação for realizada. Caso alguma operação sobre os dados viole uma condição, uma ação reparadora é realizada, evitando assim, inconsistências dos dados.

As restrições de integridade são armazenadas no catálogo do SGBD, que é o responsável pelo monitoramento das operações do usuário para garantir que as restrições não sejam violadas.

2.1.1 - TIPOS DE RESTRIÇÕES DE INTEGRIDADE

Em Elmasri [12] as restrições de integridade são classificadas, quanto a sua especificação e a sua consideração (Figura 2.1), nas seguintes categorias:

- **Restrições Implícitas**. Estão expressas no esquema do BD que está sendo modelado. Cada modelo de dados tem seu conjunto de restrições implícitas, que podem ser diretamente representadas em seus esquemas;
- **Restrições Explícitas**. São restrições que não podem ser descritas pelas restrições implícitas do modelo utilizado, necessitando de uma outro forma para serem mantidas (uma linguagem, regras, nos procedimentos das aplicações);
- **Restrições de Estado**. São aplicadas a um estado particular do BD e devem ser obedecidas por todos os dados que não estejam sendo modificados. Toda vez que uma operação modificar o BD, as restrições de estado devem ser verificadas sobre os dados modificados. Por exemplo, restrições de chave que controla a unicidade de identificadores;

- **Restrições de Transição**. São aplicadas sobre a transição dos dados no BD durante uma operação de modificação, ou seja, elas não são aplicadas sobre um estado antes ou depois da operação, mas durante a realização da operação. Por exemplo, o salário dos empregados não pode diminuir.

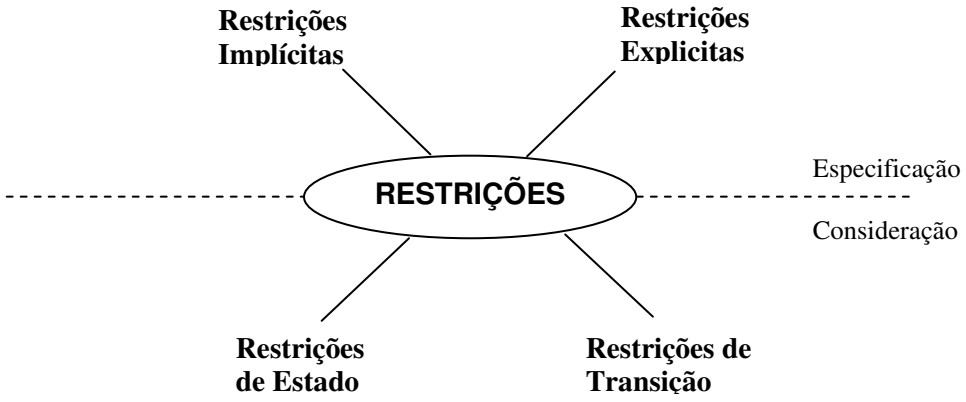

Figura 2.1 Classificação das Restrições de Integridade

Quando a especificação podemos constatar que temos dois tipos de restrições, implícitas e explícitas. Cada um destes tipos de restrições são especificadas e verificadas de formas diferentes. A seguir descrevemos, para cada tipo de restrição, a forma como é feita sua especificação.

2.1.2 - ESPECIFICAÇÃO DE RESTRIÇÕES IMPLÍCITAS

As restrições implícitas são especificadas diretamente no esquema do BD através da própria linguagens de definições de dados (DDL) do modelo utilizado. Sua verificação é implementada diretamente no SGBD e é transparente ao usuário, ou seja, o usuário apenas é notificado que uma operação não pode ser realizada pois não obedece uma determinada condição mas não sabe como a verificação foi realizada.

2.1.3 - ESPECIFICAÇÃO DE RESTRIÇÕES EXPLÍCITAS

Diferente das restrições implícitas, as restrições explícitas não podem ser especificadas através de DDL, pois estão além do seu escopo. Para sua especificação, métodos diferentes podem ser utilizados: especificação procedural, especificação declarativa como *assertions* ou utilizando *triggers*. A seguir descrevemos cada uma das alternativas.

a) ESPECIFICAÇÃO PROCEDURAL EM TRANSAÇÕES

Na especificação procedural, comandos para o controle das restrições são incluídos diretamente nas operações de atualização do BD. A vantagem desse método é que, restrições são especificadas diretamente nas operações apropriadas, garantindo sua verificação no momento certo e evitando operações desnecessárias. Mas, a especificação direta faz com que o programador tenha que conhecer, implementar e garantir a verificação das restrições necessárias. Caso haja um mau entendimento, um erro ou omissão de alguma restrição, inconsistências no BD não serão evitadas.

Outro problema é que, restrições podem mudar com o tempo. Portanto, operações afetadas pelas mudanças precisarão ser modificadas, levando aos problemas citados no parágrafo anterior.

b) ESPECIFICAÇÃO DECLARATIVA COMO ASSERÇÕES

Este método especifica de forma declarativa, todas as restrições de um BD. Asserções são especificadas, compiladas e armazenadas no catálogo do BD, de forma que podem ser referenciadas e controladas pelo SGBD. Por exemplo, para a restrição "O salário de um empregado não pode ser maior que o salário do gerente do departamento que o empregado trabalha", sua declaração em SQL pode ser:

ASSERT restrição_salario ON Empregado E G, Departamento D:

E.salario < G.salario AND D.Dept = E.Dept AND E.gerente = D.gerente and G.nome = D.Gerente

Utilizando esse método, as restrições podem ser modificadas conforme a necessidade e as transações passam a ser independentes, eliminando assim, os problemas do método procedural. Sua desvantagem é que os SGBDs não implementam este mecanismo e, os que o fazem, não o implementa forma eficiente.

c) TRIGGERS (Gatilhos)

Existem casos onde se deseja especificar uma ação, em resposta a violação de restrição. Neste caso, utilizamos *triggers* para especificar as restrições.

Triggers são formados por um evento, uma condição e uma ação. O evento determina o momento do *trigger* ser disparado. A condição determina uma situação para executar a ação. A ação é um procedimento que será executado se a condição for verdadeira.

Nos *triggers* temos a combinação dos dois métodos descritos anteriormente, onde a condição é declarativa enquanto a ação é procedural.

2.1.4 - MODELOS DE DADOS E SUAS RESTRIÇÕES DE INTEGRIDADE

Cada modelo de dados possui um conjunto de conceitos ou construtores que são utilizados para modelar uma aplicação. Agregado aos conceitos, existe um conjunto de regras que determinam a utilização correta destes conceitos. Estas regras são conhecidas como restrições de integridade.

A seguir, descrevemos as restrições de integridade de alguns modelos de dados.

a) Modelo Hierárquico

Os conceitos básicos no Modelo Hierárquico são: registros e relacionamentos tipo pai-filho, ou seja, relacionamentos 1:N entre dois registros. Os dados são organizados segundo uma coleção de árvores, formadas por registros e relacionamentos pais-filhos.

No modelo hierárquico, existem algumas restrições inerentes ao modelo, são elas:

- Todo registro em uma hierarquia deve ter um registro pai, exceto para os registros raízes;
- Se um registro filho tem dois ou mais registros pai de um mesmo tipo, então o registro filho deve ser duplicado para cada registro pai;
- Um registro filho com dois ou mais registros pais de tipos diferentes, apenas um registro pai deve ser o pai real, os outros são pais virtuais.

b) Modelo de Redes

O Modelo de Redes é baseado em dois conceitos: registros e conjuntos.

Os dados são armazenados nos registros, enquanto que os conjuntos são utilizados para modelar relacionamentos 1-N entre um registro denominado "dono" e vários registros de outro tipo, denominados "membros". Os relacionamentos são estabelecidos através de ligações (ponteiros), formando uma coleção de grafos arbitrários.

O modelo oferece restrições que são especificadas sobre registros para determinar seu comportamento, quando um novo registro é incluído ou quando um registro do tipo dono ou do tipo membro é excluído.

Restrições de inclusão determinam o que acontece quando um novo registro é inserido no BD. Neste caso temos duas opções:

- A*utomatic,* indicando que o novo registro deve ser automaticamente inserido ao conjunto apropriado;
- *Manual,* o novo registro não é inserido a nenhum conjunto. A sua inclusão é feita explicitamente pelo programador.

Restrições de Retenção são utilizadas para especificar se um registro pode existir no BD por si próprio ou tem que está relacionado a um conjunto de registros. Para as restrições de retenção, temos três opções:

- *Optional,* um registro pode existir sem esta associado a um conjunto;
- *Mandatory,* um registro tem que ser membro de um conjunto;

- *Fixed,* similar a restrição *mandatory* com a diferença de que um registro, ao ser conectado a um conjunto, não pode ser conectado a outro.

c) Modelo Relacional

O Modelo Relacional representa os dados através de relações, formadas por conjuntos de *tuplas*[1], que por sua vez, são compostas de atributos.

No modelo relacional temos três tipos de restrições de integridade:

- **Integridade de Chave.** Mantém a identificação única de uma tupla dentro de uma relação.
- **Integridade de Entidade.** Determinada que nenhum valor de chave primária poderá ser nulo.
- **Integridade Referencial.** É especificada entre duas relações para manter a consistência entre as *tuplas* das duas tabelas. A restrição referencial determina que, em um conjunto de atributos de uma relação R1 que contém valores com o mesmo domínio de um conjunto de atributos que forma a chave primária de uma relação R2, não pode aparecer nenhum valor em R1 que não esteja em R2.

d) Modelo de Entidade-Relacionamento

O Modelo de Entidade e Relacionamento [7] modela o BD através dos conceitos de entidade e relacionamento.

O modelo inclui como restrições de integridade

- Restrição de chave. Determina que cada entidade tem seu conjunto identificador de atributos, que é único entre as entidades;

- Restrições estruturais de relacionamentos. Limitam as possíveis combinações das entidades que participam de um relacionamento. Elas podem ser de dois tipos: restrição de cardinalidade, que especifica o número de instâncias de uma entidade que podem participar de um relacionamento, e restrição de participação (total

[1] Termo utilizado no Modelo Relacional para referenciar uma linha de uma tabela

ou parcial), que determina se a existência de uma entidade depende de seu relacionamento com outra entidade.

2.1.5 – RESTRIÇÕES DE INTEGRIDADE NO PADRÃO SQL-92

No padrão SQL-92 [8] temos um conjunto de restrições que podem ser classificadas em três categorias [6]: restrições de domínio, restrições de integridade referencial e asserções.

a) Restrições de Domínio

São restrições implícitas utilizadas para garantir que os dados inseridos em uma coluna devem pertencer ao um determinado domínio – conjunto de valores válidos para uma determinada coluna. As restrições de domínio são especificadas no momento de criação da tabela. São elas:

- **NOT NULL**

Determina que o valor de uma coluna não pode ser omitido;

- **UNIQUE**

Determina que em uma tabela, não pode existir duas ou mais tuplas com valores idênticos para uma determinada coluna;

- **CHECK**

Determina uma condição que deve ser respeitado por todos os valores de uma coluna;

- **PRIMARY KEY**

Determina um conjunto de colunas em uma tabela, onde seus valores são NOT NULL e UNIQUE.

A seguir, mostramos um exemplo de um comando para a criação de uma tabela, onde são utilizados algumas das restrições de domínio

```
CREATE TABLE Pessoa
(matricula CHARACTER(30) PRIMARY KEY,
  nome CHARACTER(50) NOT NULL,
  identidade CHARACTER(12) UNIQUE  );
```

Na definição da tabela "Pessoa" determinamos que, a coluna "Matrícula" não pode ser omitido e para cada tupla seu valor de ser único. Para a coluna "Nome", determinamos que seu valor não pode omitido e para a coluna "Identidade" seu valor deve único para cada tupla da tabela.

b) Restrições de Integridade Referencial

O padrão SQL permite a especificação da integridade referencial de acordo com a descrição da seção 2.3 para o modelo relacional.

Sua especificação é feita através da cláusula **FOREIGN KEY**, inseridas como parte da definição de uma tabela e indicando que uma ou mais colunas da tabela que esta sendo criada, tem uma relação de correspondência com colunas de outra tabela.

Por exemplo, considere a definição na tabela Projeto_Alocação que mantém informações a respeito dos empregados que trabalham em um determinado projeto.

```
CREATE TABLE Projeto_Alocação (
Matr_Emp CHARACTER(30) FOREIGN KEY(Matrícula) REFERENCES Empregado,
ProjetoID  CHARACTER(30) FOREIGN KEY (Cód_Projeto) REFERENCES
Projeto )
```

Nesta tabela temos a especificação de duas restrições de integridade referencial, uma entre Matr_Empr e Matrícula e outra entre ProjetoID e Cód-Projeto. Com estas restrições, determinamos que valores para Matr_Empr e ProjetoID só serão válidos se existirem valores iguais nas colunas Matrícula e Cód_Projeto, respectivamente, das tabelas referenciadas, que são Empregado e Projeto.

Uma restrição de integridade é implementada como uma regra ECA implícita do tipo

```
<foreign Key action> ::= <event><action>
<event> ::= ON UPDATE | ON DELETE
<action> ::= CASCADE | SET DEFAULT | SET NULL | NO ACTION
```

onde, cada <action> responderá de forma diferente a uma operação de exclusão ou modificação.

Por exemplo, considere que temos duas tabelas T1 e T2, onde uma coluna T2.C2 de T2 faz referencia a coluna T1.C1 de T1, ou seja, T2.C2 é uma chave estrangeira de T2 em relação a T1.C1, chave primária de T1.

CASCADE: Se ocorrer uma modificação em T1.C1 a modificação será propagada para a coluna correspondente de T2, ou seja, T1.C1=T2.C2. Já se ocorrer a exclusão de uma tupla de T1, as tuplas de T2, onde T2.C2=T1.C1 também serão excluídas;

SET DEFAULT: Atribui um valor "*default*" para as colunas T2.C2 que tiveram sua correspondente em T1, ou seja T1.C1, modificada ou excluída;

SET NULL: Atribui o valor NULL para as colunas T2.C2 que não possuem correspondentes em T1.C1;

NO ACTION: Não permite a exclusão ou modificação de uma tupla de T1 que tem sua coluna T1.C1 coincidindo com alguma com alguma coluna T2.C2 de T2;

C) Asserções

São restrições gerais que podem envolver várias tabelas. Sua sintaxe é:

<Assertion> ::= CREATE ASSERTION <nome>
CHECK (<condição>)
[constraint evaluation]

<constraint evaluation> :: = [NOT] DEFERRABLE
[{INITIALLY DEFERRED |
INITIALLY IMMEDIATE}]

A cláusula CHECK é uma condição que determina quando a asserção é ou não satisfeita. Dependendo do resultado da avaliação da condição e da forma como a avaliação é definida podemos ter como resultado: o cancelamento de um comando SQL ou o cancelamento de uma transação.

A verificação de uma restrição de integridade é realizada como uma reação a tentativa de execução uma operação de modificação sobre o BD. Este efeito reativo que o controle de integridade necessita é implementado de forma "natural" em um categoria de sistemas de bancos de dados denominada de Bancos de Dados Ativos.

2.2 - BANCOS DE DADOS ATIVOS

Bancos de Dados Ativos são sistemas capazes de detectar a ocorrência de eventos, monitorar condições específicas sobre o estado do banco de dados e executar ações sem a necessidade de intervenções externas [6, 30, 32].

Sistemas de Bancos de Dados Ativos são baseados no conceito de regras, que especificam o comportamento esperado pelo sistema diante de determinadas situações. De forma genérica, uma regra consiste de três partes: evento, condição e ação.

2.2.1 - EVENTO

Um evento consiste de um fato que ocorre em um dado momento. Eventos podem ser primitivos ou compostos e são monitorados pelo sistema para determinar o momento de disparar as regras a eles associadas.

Um evento primitivo pode ser de três tipos básicos:

- Operações sobre o banco de dados. Operações de definição ou manipulação de dados (inclusão, modificação, exclusão ou recuperação) em BD's relacionais. Em BD's orientados a objetos, um evento pode ser a criação, exclusão, modificação de um objeto ou chamada a um método que modifica o estado de um objeto;

- Eventos Temporais. Podem ser absolutos, relativos ou periódicos. Eventos temporais absolutos especificam pontos no tempo (Ex. Meio-dia, 30/05/2000), um evento temporal relativo especifica um intervalo de tempo em relação a detecção de um evento não temporal (Ex. dez minutos depois da chegada do trem) e eventos temporais periódicos são aqueles que ocorrem em intervalos regulares de tempo (Ex. Todo os dias ao meio-dia);

- Eventos Externos. São eventos definidos por aplicações externas cuja detecção é sinalizada por agentes externos (Ex. Sobrecarga de energia).

Além dos eventos primitivos, os BDA's suportam eventos compostos, ou seja, combinações de eventos primitivos através de operadores de disjunção (E1 ou E2), conjunção (E1 e E2) ou seqüência (E1; E2).

2.2.2 - CONDIÇÃO

Uma condição especifica um predicado a ser avaliado antes da execução da ação da regra.

Uma condição pode ser: um predicado simples, um predicado composto por operações lógicas em relação ao estado do banco de dados (BD) ou uma chamada a um procedimento, que pode ou não acessar o banco de dados, escrito em uma linguagem de programação de propósito geral

A especificação de uma condição é opcional. Uma regra sem condição indica que a sua ação será executada quando o evento associado for detectado e a regra for selecionada.

2.2.3 - AÇÃO

Uma ação é executada quando o evento de uma regra ocorre e sua condição é verdadeira.

Ações podem ser: operações sobre o banco de dados ou procedimentos escritos em linguagens de programação de propósito geral.

2.2.4 – VALORES DE TRANSIÇÃO

Valores de transição referem-se ao estado do BD que causou o disparo de uma regra. Basicamente, existem os seguintes mecanismos para referenciar valores de transição:

- **Evento Parametrizado.** Quando um evento parametrizado ocorre, os valores dos parâmetros podem ser referenciados pela condição ou ação da regra. Por exemplo, um evento disparado pela chamada a um método pode passar como parâmetro o identificador de um objeto que chamou um método.
- **Valores de Transição Explícitos.** Utilizam palavras reservadas para referenciar valores de transição. Por exemplo NEW, UPDATED, DELETED;

- **Valores de Transição Implícitos.** Quando uma regra é disparada por mudanças em um conjunto de dados D, então referências a D na condição ou ação, implicitamente referenciam o valor de transição associado a D.

2.2.5 - SEMÂNTICA DE EXECUÇÃO DE REGRAS

Uma vez definido o conjunto de regras em um BDA, a semântica de execução de regras pode ser expressa, de forma simplificada, pelo algoritmo da figura 2.2.

Enquanto houver regras disparadas:

1. *Selecionar uma regra disparada R*
2. *Avaliar a condição de R*
3. *Se a condição de R for verdadeira*
   Então executar ação de R

Figura 2.2 Um algoritmo de processamento de regras.

Outro aspecto relacionado à execução de regras é a interação entre o seu processamento, as operações normais de manipulação do banco de dados e o processamento de transações.

Tratando deste aspecto temos o conceito de granularidade de processamento, que especifica os pontos de execução das regras em relação às operações no BD, e o conceito de modos de acoplamento, que especifica o relacionamento entre o evento disparador e a avaliação da condição ou entre a avaliação da condição e a execução da ação.

GRANULARIDADE DE EXECUÇÃO

A granularidade de execução de regras especifica a freqüência com que uma regra é processada em relação à execução das operações de mudança de estado do banco de dados, podendo ser orientada a instância ou orientada a conjunto.

O processamento é orientado a instância se a regra é disparada para cada instância do BD que dispare ou satisfaça a condição da regra. Já o processamento

orientado a conjunto considera que, a regra será disparada uma única vez para todas as instâncias que disparem ou satisfaçam a condição da regra.

Por exemplo, considerando uma regra R que é disparada toda vez que uma nova linha é incluída em uma tabela T. Supondo que um comando é executado para incluir várias linhas na tabela T. Se a regra R for orientada a instância, para cada linha incluída a regra será disparada, mas, por outro lado, se a regra R for orientada a conjunto, ela será disparada uma única vez para o conjunto de linhas inseridas na tabela.

MODOS DE ACOPLAMENTO

Os modos de acoplamento estabelecem o relacionamento transacional entre: a ocorrência do evento e a avaliação da condição ou entre a avaliação da condição e a execução da ação.

Os modos de acoplamento podem ser:

Imediato. A avaliação da condição é realizada imediatamente depois da ocorrência do evento. Em tratando-se do par condição-ação (CA), a ação é executada logo após a avaliação da condição como verdadeira;

Retardado. A avaliação da condição é adiada até o momento antes do término da transação (antes do COMMIT), onde ocorreu o evento disparador. O mesmo ocorre entre a execução da ação e a avaliação da condição da regra;

Desacoplado dependente. A avaliação da condição só ocorre depois que a transação, onde ocorreu o evento, terminar com sucesso (COMMIT), portanto se a transação não terminar a avaliação da regra não é realizada;

Desacoplado independente. A avaliação da condição é feita em uma transação separada e é realizada independentemente do término ou não da transação onde ocorre o evento.

O processamento do par CA, em relação aos modos desacoplados, é idêntico, nos dois casos, ao processamento do par evento-condição (EC).

CONFLITO DE REGRAS

Um mesmo evento pode ser especificado para diferentes regras, causando assim, disparos simultâneos. Então, qual regra processar primeiro?

Soluções, para este problema, podem ser:

- Escolher arbitrariamente uma regra;
- Uma regra é escolhida de acordo com propriedades atribuídas no momento de sua criação;
- Uma regra pode ser escolhida de acordo com propriedades estáticas da regra, o momento de criação da regra, por exemplo;
- Através de propriedades dinâmicas atribuídas as regras, regra disparada mais recentemente, por exemplo.

A utilidade dos BDA's para as aplicações modernas é evidente, pois à medida que as aplicações vão ficando mais complexas, surge a necessidade de transferir, para o controle do sistema, tarefas que antes eram realizadas diretamente pelos projetistas de aplicações, aliviando assim, a carga de trabalho sobre os desenvolvedores e diminuindo a probabilidade de erros nos projetos. Por exemplo, controle de integridade de dados é uma tarefa onde os BDA's são bem adequados.

TERMINAÇÃO

Ações de regras podem disparar novas regras gerando assim um *loop,* muitas vez infinito. Portanto, é necessário definir um ponto de parada para o algoritmo de processamento de regras. Se a linguagem de definição de regras e as definições das regras existentes garantirem que não haverá a formação de *loops* infinitos de execução de regras, então a preocupação com a parada do processamento das regras pode ser eliminada. Mas, se a garantia não existir, deve-se determinar o ponto de parada do algoritmo, que pode ser a limitação do número de regras que podem ser disparadas após o início do processamento das regras. Nesse caso, se o número máximo for ultrapassado o programa é interrompido.

3 - TEMPORAL OBJECT MODEL (TOM)

O Modelo de Objetos Temporais – TOM [9, 24, 27] é um modelo de dados orientado a objetos com tempo e suporte a versões, que herdou sua estrutura do THM (Temporal Hierarchic Model) [13, 23].

Além das características de um modelo orientado a objetos, o TOM possibilita associar regras de integridade aos métodos através de pré-condições e pós-condições, permite a declaração de regras, considera objetos e relacionamentos temporais, versionamento de objetos e a filosofia de um modelo de dados aberto, ou seja, um modelo que permite incluir novos conceitos, possibilitando a modelagem de problemas específicos.

O TOM é composto por um conjunto predefinido de conceitos são eles: classes, relacionamento, generalização, agregação, agrupamento, herança seletiva, entre outros que descrevemos em detalhes na seção 3.1.

Em relação ao controle de integridade do modelo, o TOM disponibiliza um conjunto de regras, especificadas através de Axiomas Dinâmicos (ADs) e Efeitos Colaterais (ECs), que são utilizadas para manter a semântica estrutural dos conceitos que formam o modelo. Uma descrição detalhada dos ADs e ECs é mostrada na seção 3.3.2.

Nas próximas seções descrevemos os conceitos básicos do TOM.

3.1 - ELEMENTOS DO TOM

3.1.1 - CLASSE

Uma classe é uma coleção de objetos com estrutura semelhante. As classes podem ser:

- **Classes primitivas.** Descrevem objetos chamados de imprimíveis, que são identificados por tipos de dados primitivos (inteiro, real, string, etc.);

- **Classes Estruturadas**. Correspondem aos tipos abstratos dos modelos semânticos e suas instâncias são identificadas por um ou mais relacionamentos.

Classes podem ainda, ser classificadas como temporais ou não, dependendo se o sistema deve manter ou não o histórico de suas instâncias.

3.1.2 - RELACIONAMENTO

Um relacionamento representa uma informação sobre dois objetos que possuem uma associação válida. Cada relacionamento, no TOM, tem uma cardinalidade expressa por um par (max, min), indicando o mínimo e o máximo de instâncias que podem está associadas. Outra característica é que cada relacionamento tem um relacionamento inverso correspondente.

Os relacionamentos podem ser de dois tipos:

- **Relacionamento Instância**. É aquele que associa instâncias de duas classes;
- **Relacionamento Classe**. É aquele que relaciona uma classe como um todo a uma instância de outra classe.

Relacionamentos podem ser declarados entre duas classes estruturadas ou uma classe estruturada e uma primitiva. Este último tipo de relacionamento corresponde à noção de atributo em outros modelos de dados.

3.1.3 - ABSTRAÇÕES HIERÁRQUICAS

Uma abstração hierárquica consiste em criar novas classes a partir de classes já existentes, abstraindo detalhes e acrescentando propriedades específicas às novas classes.

O TOM suporta como hierarquias de classes: generalização/especialização, agregação e agrupamento.

GENERALIZAÇÃO/ESPECIALIZAÇÃO

Uma Generalização/Especialização significa a criação de uma classe genérica para descrever propriedades comuns a instâncias de diversas classes existentes. As classes mais específicas irão herdar estas propriedades comuns e manter somente a descrição das propriedades específicas. Em resumo, podemos afirmar que, a generalização é um relacionamento entre uma classe (superclasse) e uma ou mais versões refinadas (subclasses) da mesma classe.

A toda generalização está associado um predicado que determina a que subclasse cada instância da superclasse irá pertencer.

No TOM temos os seguintes tipos de generalização/especialização:

- **Especialização Disjunta.** Ocorre quando cada objeto da superclasse ocorre no máximo em uma subclasse;
- **Especialização Completa.** Ocorre quando toda instância da superclasse for instância de pelo menos uma subclasse;
- **Especialização por Predicado**. Quando o critério de especialização dos objetos pode ser dado por uma condição;
- **Especialização Explicita**. Ocorre quando, ao incluirmos um objeto na superclasse, é necessário especificar, explicitamente, a que subclasse o objeto pertencerá.

AGREGAÇÃO

Consiste em criar objetos compostos a partir de outros objetos. Em outras palavras, em uma agregação os objetos da classe agregada são formados pela combinação de objetos das classes componentes. Cada objeto componente é chamado de parte-do objeto composto.

No TOM temos os seguintes tipos de agregação:

- **Agregação Total.** Ocorre quando a classe agregada é formada por todo o produto cartesiano das classes componentes;
- **Agregação por Relacionamento**. Ocorre quando os objetos agregados são obtidos por um relacionamento entre as classes componentes;

- **Agregação Exclusiva**. Ocorre quando cada componente não pode participar em mais de uma agregação;

AGRUPAMENTO

Consiste em criar uma classe formada por um conjunto de instâncias de outra classe. Cada objeto que participa de um agrupamento é chamado de elemento-do objeto grupo. A classe que fornece os elementos dos grupos é chamada de classe-elemento e a que contém os grupos é a classe-grupo.

No TOM temos as seguintes categorias de agrupamento:

- **Agrupamento Total.** Ocorre quando todas as instâncias da classe-elemento pertencem à classe-grupo;
- **Agrupamento Explícito.** Ocorre quando a estratégia de agrupamento é determinada explicitamente pelo usuário;
- **Agrupamento Implícito.** Ocorre quando a estratégia de agrupamento é determinada por um predicado;
- **Agrupamento Disjunto.** Ocorre quando uma instância da classe-elemento pode pertencer, no máximo, a um grupo, ou seja, a uma única instância da classe-grupo;
- **Agrupamento Completo.** Ocorre quando cada instância da classe-elemento é elemento de pelo menos um grupo.

HERANÇA

Além da herança automática para a generalização, o TOM oferece um tipo de herança seletiva para agregação e agrupamento. Assim temos, nestas duas abstrações:

- **Herança Direta**. Os relacionamentos são herdados diretamente pelo objeto de nível inferior sem modificações. Por exemplo, uma instância da classe "Carro" tem um relacionamento com uma instância da classe "Dono" indicando posse. Este relacionamento é herdado pelas partes do carro sem modificações, uma vez que o dono do carro é dono de suas partes;

- **Herança Computada**. Alguns valores, que não podem ser herdados diretamente, são resultado da execução de uma operação sobre alguns relacionamentos. Por exemplo, em uma agregação o peso do carro é igual a soma dos pesos de suas partes.

3.1.4 - ASPECTOS TEMPORAIS

O TOM modela o fator Tempo através de:

- **Classes temporais.** Cada instância tem associado seu tempo de duração na mesma classe. Este tempo é representado por um ou vários intervalos temporais;
- **Relacionamentos temporais.** Os valores dos relacionamentos modificados são mantidos no banco de dados para efeitos históricos, com a indicação do intervalo válido (inicio e fim) para os valores anteriores do relacionamento.
- **Relacionamentos pré-pós.** Estes relacionamentos estabelecem uma ordem de prioridade entre classes, com relação as operações de inclusão e exclusão. Por exemplo, ao eliminarmos uma instância de uma classe A esta deve passar, obrigatoriamente, a pertencer a uma classe B, assim, temos um relacionamento A→pré→B.

3.1.5 - MÉTODO

Um método é um procedimento de mudança de estado de um objeto, que se reflete na alteração dos seus relacionamentos e/ou envio de mensagens a outros objetos.

No TOM existem dois tipos de métodos:

- **Métodos de instância**, onde a chamada ao método é realizada através de uma mensagem enviada a uma instância de uma classe;
- **Método de classe**, onde a mensagem é enviada a uma classe. É utilizado para manipular relacionamentos-classes, criar novas instâncias ou selecionar várias instâncias de uma classe.

3.1.6 – OBJETOS VERSIONADOS

Versões são várias ocorrências de um mesmo objeto, que coexistem em uma base de dados e representam diferentes estados assumidos durante sua existência.

No TOM, todo objeto com versões, possui uma parte genérica e as partes versionadas, onde, na parte genérica encontram-se informações comuns do objeto ao longo de todas as versões. As versões do objeto estão relacionadas com sua parte genérica, formado um grafo acíclico denominado **grafo de versões**.

Objetos versionados não são considerados neste trabalho.

3.2 - OPERAÇÕES ELEMENTARES

As operações elementares (Tabela 3.1) são operações predefinidas do TOM utilizadas para manipular os objetos do BD. Toda modificação em um banco de dados TOM é realizada, exclusivamente, através de uma operações elementares.

OPERAÇÕES	DESCRIÇÃO	SINTAXE
Create	Cria um novo objeto na classe especificada.	classe create (x)
Delete	Eliminar um objeto da classe especificada.	x delete (classe)
Establish	Estabelece o relacionamento r entre os objetos x_1 e x_2.	x establish (rel, y)
Remove	Remove o relacionamento r entre os objetos x_1 e x_2.	x remove (rel, y)
Update	Atualiza o relacionamento r entre o objeto x_1.	x update(rel, y1, y2)
Gr-insert	Inclui o objeto x como um novo elemento do grupo g.	grupo gr_insert(x)
Gr-delete	Elimina o objeto x do grupo g	grupo gr_delete(x)

Tabela 3.1 Operações Elementares

Associadas às operações elementares existem regras que determinam sob quais condições do BD, tais operações podem ser executadas e também um conjunto de operações elementares complementares que devem ser realizadas, sob determinadas condições, depois de cada operação elementar a fim de manter a integridade dos dados no BD.

3.3 - INTEGRIDADE SEMÂNTICA NO TOM

A integridade semântica de um BD TOM, refere-se a não violação da semântica dos conceitos que constituem o modelo e é verificada sempre que uma modificação for imposta ao BD. Exemplos de violação são: um objeto pertencer a um agrupamento sem pertencer a classe elemento (violação no conceito de agrupamento) ou um objeto está em uma subclasse sem está na superclasse (violação do conceito de generalização).

O controle de integridade é feito através dos Axiomas Dinâmicos (AD), que impedem a violação de restrições especificadas no esquema conceitual, cancelando a operação violadora, e dos Efeitos Colaterais (EC), que executam operações, complementares às operações elementares, para trazer o banco de dados a um estado que esteja de acordo com o esquema conceitual da aplicação e não viole a semântica dos conceitos utilizados.

Cada AD ou EC é descrito através de uma regra ECA que tem como evento a execução de uma operação elementar, como condição uma expressão formada de predicados estruturais – predicados que verificam os dados e as classes das aplicações segundo a estrutura dos conceitos do TOM, e como ação o cancelamento (*abort*) da operação que sinalizou o evento ou outra operação elementar.

De agora em diante, regras que representam os ADs e ECs nos referiremos no texto como regras do modelo, enquanto que, as regras que modelam restrições semânticas inerentes às aplicações chamaremos de regras de aplicação.

A respeito dos predicados estruturais, utilizados na especificação da condição dos ADs e ECs, cada conceito que compõe o modelo possui seus próprios predicados, definidos para verificar o estado e a definição do BD segundo as características estruturais do próprio conceito.

Nas próximas seções, descrevemos os predicados estruturais, os axiomas dinâmicos e efeitos colaterais de cada conceito do modelo.

3.3.1 - PREDICADOS ESTRUTURAIS

Os predicados estruturais foram especificados com base nos axiomas estáticos do TOM – axiomas que descrevem as estruturas dos conceitos do modelo.

Cada predicado é utilizado, como mencionado na seção anterior, para verificar os dados e a organização das classes do BD, segundo a estrutura do conceito a que ele pertence.

Então, utilizando os predicados estruturais nas suas condições, os ADs e ECs testam se uma determinada restrição semântica do modelo vai ser ou foi violada por alguma operação sobre o BD, cancelando a operação ou executando algum operação complementar para corrigir a violação.

A seguir, descrevemos os predicados estruturais relacionados a cada conceito do TOM.

Predicados Primitivos

Classe/Relacionamento

x in C	Verifica se o objeto x pertence a classe C.
Is_rel(x, r ,y)	Verifica se os objetos e1 e e2 estão relacionados por r.
Class_rel(r,C1,C2)	Verifica se as classe C1 e C2 estão relacionadas por r.
In_rel(C,r)	Verifica se a classe C esta envolvida no relacionamento r.
Max(x,r)	Verifica se o número de instância relacionadas a x pelo relacionamento r atingiu a cardinalidade máxima do relacionamento.
Min(x,r)	Verifica se o número de instância relacionadas a x pelo relacionamento r atingiu a cardinalidade mínima do relacionamento.
Inv(r)	Verifica o relacionamento inverso do relacionamento r.

Tabela 3.2 Predicados Primitivos

Hierarquias de Tipos

Generalização/Especialização

Is-a (C1,C2)	Verifica se a classe C1 é uma subclasse de C2.
In_subclasse(x, Ci, D)	Verifica se o objeto x pertence a uma subclasse de D diferente da subclasse Ci.

Papel (x,D,C)	Verifica o critério que deve ser satisfeito por todas as instâncias de uma subclasse C de D.
Disjuntivo (D, C1...Cn)	Verifica se D é uma generalização disjuntiva de C1...Cn.
Covering (D,C1...Cn)	Verifica se D é uma generalização covering de C1...Cn.

Tabela 3.3 Predicados da Generalização/Especialização

Agrupamento

Is-elem (x1,g)	Verifica se o objeto x1 é elemento do grupo g.
Is-group (G, C)	Verifica se a classe G é um agrupamento de elementos da classe C.
In-group(g, G)	Verifica se o elemento g pertence a classe-grupo G
Gr-disjuntivo (G, C)	Verifica se G é um agrupamento disjuntivo de elementos da classe C.
Gr-covering (G, C)	Verifica se G é um agrupamento covering de elementos da classe C.
In-another-group(x, g, G)	Verifica se o objeto x pertence um grupo da classe G diferente de g
Only-in-group(x, g, G)	Verifica se o objeto x pertence a apenas o grupo g da classe G
Gr_papel(x, g ,G)	Verifica o critério que deve ser satisfeito por todas as instâncias de um grupo

Tabela 3.4 Predicados do Agrupamento

Agregação

Is_part (e1,e2)	Verifica se o objeto e1 é componente do objeto e2.
Class_is_part(C,D)	Verifica se a classe C é componente da classe D.
Agregado (A,C1...Cn)	Verifica se a classe A é uma agregação de classes C1...Cn.
Agregado-r (A,C1,C2,r)	Verifica se a classe A é uma agregação de C1 e C2 por r.

Tabela 3.5 Predicados da Agregação

Tempo

Timed_class(C)	Verifica se uma classe C é temporal.
Timed_rel (r,C1, C2)	Verifica se r é um relacionamento com valores temporais anteriores entre C1 e C2.
Excl_pré (C, D)	Verifica se existe uma relação indicando que os objetos removidos de D são inseridos em C.
Excl_pós (C, D)	Verifica se existe uma relação entre C e D indicando que, os objetos de D, obrigatoriamente, originaram-se de C.
O(x, C, D)	Verifica se depois da modificação de um relacionamento o objeto x pode pertencer a alguma subclasse C de D.
Overlap(x, t1,t2,C)	Verifica se o objeto já existe e está sendo sobreposto pelo novo objeto

Tabela 3.6 Predicados do Tempo

Herança

Agg_herança_dir (A, C1...Cn, r)	Verifica se A é uma agregação de C1...Cn com herança direta de r.
Agg-heranca-comp(A,C1...Cn, r, θ)	Verifica se a classe A é uma agregação de C1...Cn com herança computada, ou seja, a classe agregada herda propriedades computadas das classes componentes.

Tabela 3.7 Predicados da Herança

Descritos os predicados estruturais dos conceitos do TOM, podemos descrevemos os ADs e ECs que fazem uso de tais predicados.

3.3.2 - AXIOMAS DINÂMICOS E OS EFEITOS COLATERAIS

Os axiomas dinâmicos funcionam como pré-condições que impedem a realização de operações que torne o BD inconsistente com o esquema conceitual. Por exemplo, na definição do um BD foi definido um relacionamento com uma determinada cardinalidade, então é um AD que verifica se, para um determinado objeto, a cardinalidade máxima do relacionamento foi atingida, antes de permitir que o objeto seja associado ao outro objeto pelo relacionamento.

Já os efeitos colaterais executam, implicitamente, operações adicionais para recompor a integridade semântica da base de dados, afetada em decorrência da execução de uma operação elementar. Por exemplo, em uma generalização, se for incluído um objeto em uma subclasse e este objeto não estiver na superclasse, é um EC que incluirá o novo objeto na superclasse.

A seguir, mostramos os axiomas dinâmico e os efeitos colaterais do TOM, expressando-os na forma de regras ECA [10], ou seja, ON <evento> IF <condição> DO <ação>.

AXIOMAS DINÂMICOS

AD1 ⇒ "*Um establish deve conservar a cardinalidade máxima*"

ON *x establish (r, y)* IF *is-rel (x,r,y)* $\wedge$ *max(x,r)* DO *abort*

AD2 ⇒ "*Um remove deve conservar a cardinalidade mínima*"

ON *x remove (r, y)* IF *is-rel (x,r,y)* $\wedge$ *min(x,r)* DO *abort*

AD3 $\Rightarrow$**"*Em uma generalização, a inserção de um objeto deve respeitar o papel*"**

ON *C create (x)* IF *is-a (C, D)* $\wedge$ *not papel(x,D,C)* DO a*bort*

AD4 $\Rightarrow$ **"*Em uma generalização covering, não é permitido que um objeto pertença apenas à classe genérica*"**

ON *x delete (C)* IF *is-a (C, D)* $\wedge$ *covering (D,* C_1*,...,*C_n*)* $\wedge$ *not in_subclasse(x, C, C1...Cn)* DO a*bort*

AD5 $\Rightarrow$ **"*Em uma classe com tempo, não é permitido a criação de uma instância já presente*"**

ON *C create (x)* IF *timed_class (C)* $\wedge$ *x in C* $\wedge$ *overlap (x,* t_1*,*$t_{2,C}$*)* DO *abort*

AD6 $\Rightarrow$ **"*Elementos de grupo devem obedecer ao predicado do agrupamento*"**

ON *g gr-insert (x)* IF *in-group(g,G)* $\wedge$ *not gr_papel(x,g,G)* DO a*bort*

AD7 $\Rightarrow$ **"*Para um agrupamento covering, cada objeto da classe elemento deve estar em pelo menos um grupo*"**

ON *C create (x)* IF *gr-covering (C, G)* $\wedge$ *in_group(g G)* $\wedge$ *not is_elem(x, g)* DO a*bort*

EFEITOS COLATERAIS

- Generalização

EC1 $\Rightarrow$**"*Um objeto de uma subclasse deve pertencer à superclasse*"**

ON *C create (x)* IF *is-a (C, D)* $\wedge$ *not x in D* DO *D create (x)*

EC2 $\Rightarrow$**"*Um create não deve violar uma generalização disjunta*"**

ON *C create (x)* IF *is-a (C, D)* $\wedge$ *disjuntivo (D,* C_1*,...,*C_n*)* $\wedge$ *in_subclasse(x, C, D)* DO *x delete (*C_i*)*

EC3 ⇒"*Um delete não deve violar uma generalização covering*"

ON *x delete (C)* IF *is-a (C, D)* ∧ *covering (D, C_1,...,C_n)* ∧ *not in_subclasse(x, C, D)* DO *x delete (D)*

EC4 ⇒ "*Se, como conseqüência da alteração de um relacionamento, não é permitido a um objeto permanecer numa subclasse, ele deve ser movido para uma subclasse compatível*"

ON *x establish (r, y)* or *x remove(r, y)* IF *in_rel(C, r)* ∧ *x in C* ∧ *is_a (C, D)* ∧ *papel(x, C, D)* ∧ *not o(x,C, D)* DO *x delete(C)*

ON *x remove (r, y)* or *x establish (r, y)* IF *in_rel(C, r)* ∧ *x in C* ∧ *is_a (C, D)* ∧ *not papel(x, C, D)* ∧ *o(x, C, D)* DO *D create (x)*

EC5 ⇒ "*Um novo objeto de uma classe genérica deve ser inserido em todas as subclasses*"

ON *D create (x)* IF *is-a (C, D)* ∧ *papel(x, C, D)* DO *C create (x)*

EC6 ⇒ "*Ao ser eliminado um objeto de uma classe genérica, ele deve ser eliminado em todas as subclasses*"

ON *x delete (D)* IF *is-a (C, D)* ∧ *in_subclasse(x, A, C)* DO *x delete (C)*

- Agregação

EC7 ⇒ *Se um objeto agregado ficar sem um de seus componentes, ele deve ser eliminado*"

ON *x delete (C)* IF *class-is-part (C, D)* ∧ is-part*(x,y)* ∧ *y in D* DO *y delete (D)*

EC8 ⇒ "*Para uma agregação por relacionamentos, a remoção de um relacionamento provoca a eliminação do agregado correspondente*"

ON *x remove (r, y)* IF *agregado-r (D, C_1,C_2, r)* ∧ DO *<x,y> delete (D)*

$x \text{ in } C_1 \wedge y \text{ in } C_2$

EC9 ⇒ "*Para uma agregação por relacionamento, a inserção de um relacionamento provoca a inserção do objeto agregado correspondente*"

ON *r establish(x,y)* IF *agregado-r (D, C_1,C_2, r)* ∧ *x in C_1* ∧ *y in C_2* DO *D create (<x,y>)*

EC10 ⇒"*As partes de um objeto agregado devem pertencer às classes componentes*"

ON *D create (y)* IF *agregado (D, C_1,...,C_n)* ∧ *is-part (x_i, y)* DO *C_i create (x_i)*

EC11 ⇒*"As partes de um objeto agregado devem pertencer às classes componentes e os relacionamentos correspondentes devem ser estabelecidos"*

ON *D create (y)* IF *agregado-r (D, C_1,C_2, r)* ∧ class_*rel (r,C_1,C_2)* ∧ *is-part (x_1, y)* ∧ *is-part (x_2, y)* DO *D C_1 create (x_1)* *C_2 create (x_2)* *x1 establish(r, x_2)*

- Agrupamento

EC12 ⇒ "*Se p(x,g G) é satisfeito, então se x é inserido na classe elemento, deverá ser acrescido a g*"

ON *C create (x)* IF *is-group (G,C)* ∧ *in_group(g,G)* ∧ *gr_papel(x, g, G)* DO *g gr-insert (x)*

EC13 ⇒"*Por um agrupamento covering, se o objeto for eliminado do grupo, ele deve ser eliminado da classe elemento*"

ON *x gr-delete(g)* IF *in_group(g,G)* ∧ *gr-covering (C, G)* ∧ *x in C* DO *x delete (C)*

EC14 ⇒"*Se o objeto da classe elemento for eliminado, ele deve ser eliminado do grupo*"

ON *x delete (C)* IF *is-group(G,C)* ∧ *instancia_grupo(g,G)* ∧ is_*elem(x,g)* DO *x gr-delete (g)*

EC15 ⇒"*Os elementos de um grupo devem estar na classe elemento*"

ON x *gr_insert (g)* IF *in_group(g,G)* ∧ is_group(G,C) ∧ not x in C DO *C create (x)*

EC16 ⇒"*Em um agrupamento covering, se for eliminado um grupo, os elementos deste grupo são eliminados na classe elemento*"

ON *g delete (G)* IF *only_in_group (g, G) ∧ gr-covering (C, G)* DO *x delete (C)*

EC17 ⇒"*Se a inserção de um objeto no grupo violar a propriedade da disjunção, o objeto é removido dos outros grupos*"

ON *g gr-insert (x)* IF *in_group(g,G) ∧ gr_disjuntivo (C, G) ∧ in_another_group(x,g,G,h)* DO *x gr-delete(h)*

- Relacionamento

EC18 ⇒"*Um delete provoca a remoção de todos os relacionamentos existentes*"

ON *x delete (C)* IF *is-rel (x,r,y) ∧ in_rel(C,r)* DO x *remove (r,y)*

EC19 ⇒"*Toda a relação tem um inverso*"

ON *x establish (r, y)* IF *not is-rel(y,inv(r),x)* DO *y establish(inv(r),x)*

- Aspectos Temporais

EC20 ⇒"*Ao ser criado um objeto temporal, deve ser registrado o instante da sua criação*"

ON *C create (x)* IF *timed_class (C)* DO *C create_temporal(x)*

EC21 ⇒"*Em uma classe com tempo, um objeto eliminado é movido para o passado*"

ON *x delete (C)* IF *timed_class (C)* DO *C delete_temporal(x)*

EC22 ⇒"*Na criação de um relacionamento temporal, o tempo implícito deve ser criado*"

ON x *establish (r, y)* IF *timed_rel(r)* DO *establish_rel_temporal(x,r,y,t1,now)*

EC23 ⇒"*Na remoção de um relacionamento temporal, ele deve ser removido para o passado*"

ON *x remove (r,y)* IF *timed_rel(r)* DO *remove_rel_temporal(x,r,y,t1,t2)*

EC24 ⇒"*Em uma relação pré-pós, a inserção de um objeto em uma pós-classe, elimina este objeto na pré-classe*"

ON *D create (x)* IF *excl-pre (C, D) ∧ x in C∧ not timed(C)* DO *x delete (C)*

EC25 ⇒"*Em uma relação pré-pós, a eliminação de um objeto em uma pré-classe, insere este objeto na pós-classe*"

ON *x delete (C)* IF *excl-pos (C, D) ∧ not x in D* DO *D create (x)*

- Herança

EC26 ⇒"*Em uma agregação com herança direta, a alteração no relacionamento do objeto agregado x implica na alteração do relacionamento do objetos componentes z* "

ON x *update (r,y1,y2)* IF *agg-herança(A, A_1,....,A_n, r)* DO z *update (r,y1,y2)*
∧ is-part (z, x)

EC27 ⇒"*Em uma agregação com herança computada, a alteração no relacionamento de um objeto componente z implica na alteração do relacionamento do objeto agregado x* "

ON z *update (r,y1,y2)* IF *agg-herança-comp(A, A_1,....,A_n,,r,θ)* DO *x update(r, y1, y3)*
∧ is-part (z, x) ∧ y3 = θ(y2)

3.4 - METANÍVEL DO TOM

O TOM é um modelo aberto orientado a objetos, pois dispõe de um metanível (Figura 4.3) composto por um sistema de metaclasses, que possibilita a descrição de conceitos de modelos de dados, independentes de aplicações. Ele disponibiliza uma metaclasse predefinida, denominada TOM-Class, a partir do qual podem ser criadas novas metaclasses ou classes, caracterizando assim, dois níveis de descrição: o metanível e o nível de aplicação. Modelos com mecanismos similares de metaclasses são descritos em [11, 21].

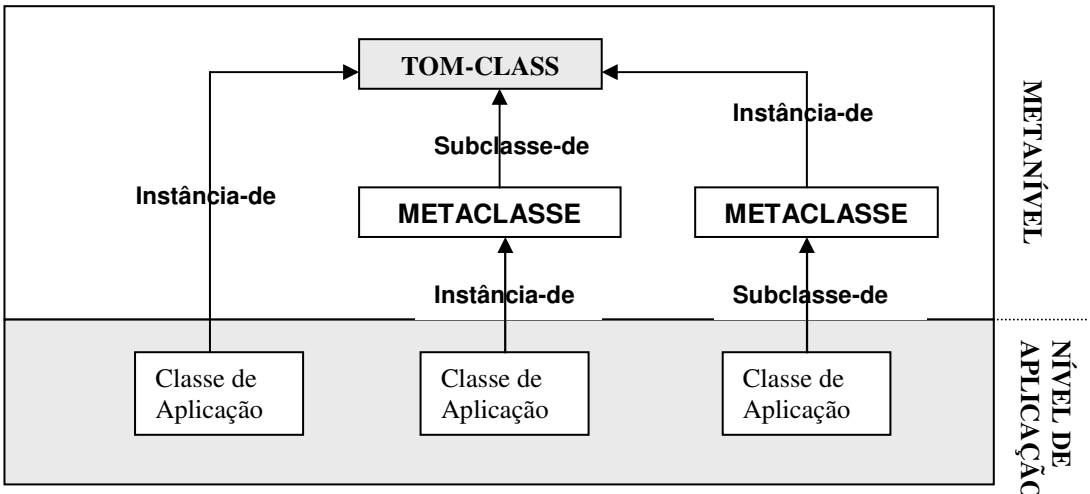

Figura 3.1 Níveis de Abstração no TOM

No metanível, o projetista de modelos pode criar novas metaclasses, como instância ou especialização da TOM-Class, disponibilizando novos conceitos para criar um modelo específico para uma ou várias aplicações. O próprio modelo TOM foi gerado a partir da TOM-Class [25].

No nível de aplicação, o projetista vai utilizar as metaclasses, para desenvolver os esquemas conceituais das aplicações, criando classes como instâncias ou especializações da TOM-Class ou das metaclasses definidas no metanível, ou seja, o projetista utiliza as metaclasses como conceitos do modelo necessários à solução de um problema específico.

Quando uma nova metaclasse é criada, como uma subclasse de TOM-Class, um novo conceito é inserido ao modelo e, junto a este novo conceito, deve ser especificado um conjunto de regras de integridade, ou seja, axiomas dinâmicos e efeitos colaterais, para garantir as características semânticas do conceito no nível de aplicação.

A especificação de novos ADs e ECs implica na definição de novos predicados estruturais e de novas regras para modelá-los. Portanto, é necessário um sistema de regras que seja capaz de processar os predicados estruturais e regras do modelo referentes aos conceitos predefinidos do TOM, como também, possa absolver os predicados estruturais e as novas regras do modelo criadas em conseqüência da definição de novos conceitos no TOM.

4 - OBJETIVO E ESPECIFICAÇÃO DO GERENCIADOR DE REGRAS ATIVAS DO TOM

4.1 - OBJETIVO DO GeRATOM

O GeRATOM é um sistema de regras ativas para controlar restrições de integridade implícitas e explícitas no TOM. O GeRATOM considera a característica aberta do modelo TOM, permitindo a inclusão de novas restrições implícitas a medida que o modelo é modificado.

As funcionalidades oferecidas pelo GeRATOM permitem:

a) A inclusão de novos conceitos, novas regras modelando os ADs e ECs do novo conceito e novos predicados estruturais definidos para especificar os novos ADs e ECs;
b) O processamento dos predicados estruturais predefinidos do modelo ou definidos pelo usuário durante a avaliação da condição de uma regra;
c) O processamento de regras parametrizadas e a troca de parâmetros entre as partes de uma regra e entre os predicados durante a avaliação de sua condição;
d) Uma maior portabilidade do sistema, pois consideramos o padrão SQL-92 na sua implementação.

As funcionalidades a), b) e c) possibilitam a especificação e o processamento dos ADs e ECs através de regras ativas e garantem a característica aberta do TOM no que se refere ao controle de integridade. A funcionalidade d) é para possibilitar portabilidade do sistema em relação a diferentes SGBDs relacionais.

4.2 - ESPECIFICAÇÃO DO GeRATOM

Nesta seção descrevemos a forma como o GeRATOM manipula os elementos referentes ao controle de integridade do TOM. Todas as informações destes elementos são mantidas em uma base de regras que será descrita em maiores detalhes no capítulo 5 e no apêndice B deste trabalho.

4.2.1 - CONCEITOS

O objetivo do GeRATOM é permitir o processamento das regras de integridade dos conceitos predefinidos e dos conceitos que venham a ser incluídos posteriormente no TOM.

Os conceitos do modelo precisam ser cadastrados no sistema para que o GeRATOM possa processar suas regras de integridade. Nesse sentido, o GeRATOM permite a manipulação de conceitos no sistema através de duas operações: incluir e excluir um conceito (Figura 4.1).

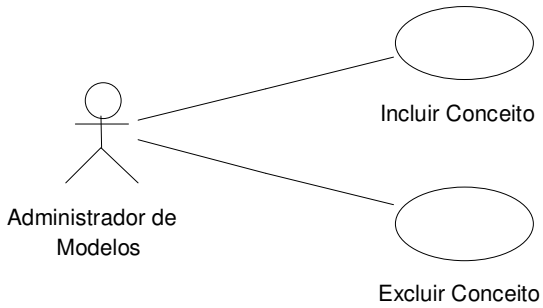

Figura 4.1 Manipulação de Conceitos do TOM

Para realizar estas operações temos os seguintes comandos:

a) **Inserir_conceito** – Inclui um novo conceito no cadastro de conceitos do sistema
   Sintaxe: Inserir_conceito (<nome_conceito>)
   Exemplo: Inserir_conceito('Generalização')

b) **Excluir_conceito** – Exclui um conceito do cadastro de conceitos
   Sintaxe: Excluir_conceito (<nome_conceito>)
   Exemplo: Excluir_conceito('Generalização')

As operações para a manipulação de conceitos são necessárias, pois, as regras definidas para modelar ADs e ECs do TOM são associadas, inicialmente, aos conceitos e não às classes das aplicações. A associação entre ADs e ECs e as classes da aplicação é feita posteriormente através de uma operação definida para este propósito.

4.2.2 - AXIOMAS DINÂMICOS E EFEITOS COLATERIAS

Cada conceito do TOM tem associado um conjunto de regras, ADs e ECs, que devem ser obedecidas para garantir a integridade dos dados no BD. O GeRATOM deve permitir a inclusão e a exclusão destes elementos (Figura 4.2).

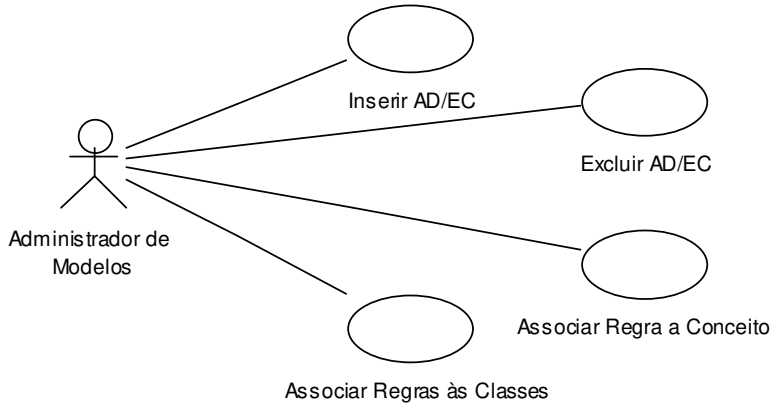

Figura 4.2 Manipulação de ADs e ECs

Os ADs e ECs são modelados e inseridos no sistema como regras ativas, então, as operações de manipulação de regras, descritas ainda neste capítulo, são válidas também para a manipulação dos ADs e ECs. As regras que modelam ADs e ECs são definidas de forma genérica, ou seja, não fazem referências à classes ou relacionamentos das aplicações. Por exemplo,

EC20 ⇒"*Ao ser criado um objeto temporal, deve ser registrado o instante da sua criação*"

ON *C create (x)* IF *timed_class (C)* DO *C create_temporal(x)*

Nesta regra podemos verificar que toda a sua definição é feita através de variáveis que, no momento do processamento da regra, receberão valores reais, neste caso, o nome da classe e o objeto criado. Então, é necessário definir operações para permitir que estas regras sejam associadas às classes ou relacionamentos das aplicações como forma de garantir que tais regras serão disparadas de forma correta e manterão a integridade dos dados adequadamente.

Duas operações foram definidas com este propósito: uma para associar uma regra a um conceito e outra para associar regras às classes e relacionamentos das aplicações

A associação entre uma regra e um conceito é feito com o objetivo de indicar que uma determinada regra representa um AD ou EC do conceito. Este relacionamento permite que, apenas analisando a definição de uma classe e identificando quais conceitos foram utilizados em sua definição, seja possível associar às classes e a seus relacionamentos, os ADs e ECs adequados.

Por exemplo, uma classe X é definida como temporal, ou seja, ela utiliza o conceito "Tempo" em sua definição. Ao analisar a definição da classe X é identificado que o conceito "Tempo" foi utilizado na sua definição, então, com esta informação, o sistema associará as regras que modelam os ADs e ECs de "Tempo" a classe X.

Assim, de acordo com a seção 3.3.2 que indica que o conceito "Tempo" no TOM possui os seguintes ADs e ECs: AD5, EC20, EC21, EC22, EC23, EC24 e EC25 e considerando que a classe X é uma classe temporal, teremos a seguinte associação entre a classe X e as regras que modelam os ADs e ECs do "Tempo".

Classe	Regras Associadas
X	AD5, EC20, EC21, EC22, EC23, EC24, EC25

Estas informações são mantidas na base de regras do GeRATOM e, a partir de tais informações, o sistema identifica as regras que devem ser disparas quando uma determinada classe ou relacionamento é modificado. Para o exemplo, se a classe X for modificada, as regras que modelam AD5, EC20, EC21, EC22, EC23, EC24 e EC25 serão disparadas.

A operação para associar uma regra a um conceito é realizada através do comando **Relacionar_regra_conceito**, cuja sintaxe é a seguinte:

Sintaxe: Relacionar_regra_conceito (<nome_regra>, <nome_conceito>).

Exemplo: Relacionar_regra_conceito ('Regra01', 'Generalização').

Neste exemplo uma regra identificada por 'Regra01' é associada ao conceito 'Generalização' do TOM, a partir desse momento, a regra 'Regra01' caracteriza um AD ou EC do conceito 'Generalização'.

Já a associação entre regras e classes/relacionamentos de aplicação é feita através do comando **Associar_regras_aplicação**

Sintaxe: Associar_regras_aplicação (<nome_BD>)

Exemplo: Associar_regras_aplicação(‘Concessionária’)

Este comando associa as regras às classes e relacionamentos do banco de dados cujo nome é ‘Concessionária’.

Com esta duas operações garantimos que os ADs e Ecs serão manipulados e processados de forma correta pelo sistema para garantir a integridade dos dados segundo as características estruturais dos conceitos do TOM. Mas, considerando que regras podem ser definidas para modelar restrições específicas a uma classe ou relacionamento, por exemplo, uma regra Mantém_salário que não permite que em uma empresa o salário de um empregado seja modificado para um valor menor que o salário atual (Figura 4.3)

ON C Update (x, tem_salario, salaria_atual, novo_salario)
IF novo_salario < salario_atual
THEN Abort

Figura 4.3. Mantém Salário

Como podemos verificar na definição da regra da figura XX, nenhuma referência à classe Empregado é feita, então como garantir que a regra será disparada apenas quando a classe Empregado for modificada, mais especificamente o relacionamento “Tem_salário”.

Com este objetivo o GeRATOM disponibiliza uma operação que permite associar uma regra diretamente a uma classe ou relacionamento de uma aplicação, criando explicitamente a associação entre classe/relacionamento e regras obtido depois da execução do comando **Associar_regras_aplicação.** Esta operação é realizada através do comando **Associar_regra_classe.**

Sintaxe: Associar_regra_classe (<nome_regra>, <nome_classe>).

Exemplo: Associar_regra_classe (‘Regra-Exemplo’, ’Tem_salário’).

Para o exemplo da figura 4.3 depois da execução da operação Associar_regra_classe teremos a seguinte associação estabelecida entre a Mantém_salário e o relacionamento ‘Tem_salário’

Classe	Regras Associadas
Tem_salário	Mantém_salário

E de mesma forma dos AD's e EC's o sistema poderá identificar qual regra deverá ser disparada caso o relacionamento 'Tem_salário' seja modificado.

4.2.3 - PREDICADOS ESTRUTURAIS

Todos os conceitos do TOM possuem um conjunto de predicados estruturais que verificam os dados do BD, segundo as características estruturais dos conceitos. Os predicados estruturais são os elementos básicos para a especificação dos ADs e ECs, pois suas condições são definidas utilizando tais predicados.

Então, agregada às funcionalidades para a inclusão de novos conceitos e de novos ADs e ECs, o GeRATOM deve permitir a inclusão de predicados estruturais (Figura 4.4), possibilitando assim, a especificação dos ADs e ECs dos conceitos.

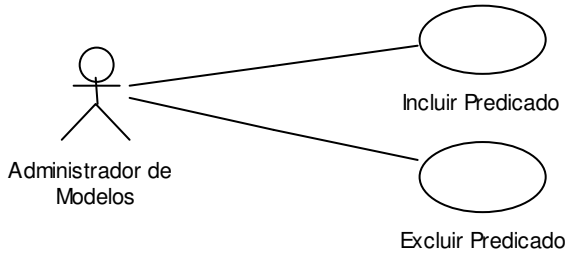

Figura 4.4 Manipulação de Predicados Estruturais

Os predicados estruturais são funções que acessam o BD sendo implementados em uma linguagem de programação suportada pelo SGBD base. Inicialmente, as funções que implementam predicados não têm nenhuma ligação com os conceitos do modelo, apenas quando uma destas funções é referenciada em um AD ou EC de um conceito a função é considerada como um predicado estrutural do conceito.

Além das referências nas regras, é necessário fornecer ao GeRATOM meios para que ele possa identificar os parâmetros que devem ser passados para estas funções. Estas informações são indispensáveis para o processamento de um predicado durante a avaliação da condição de um AD ou EC e são fornecidas durante a execução da operação de inclusão de um predicado no GeRATOM que é realizada através da execução do comando **Incluir_Predicado.**

Sintaxe: Incluir_Predicado (<nome_predicado> , {<Parâmetros>}).

<Parâmetros> ::= <nome : <tipo>>; **...** ;<nome : <tipo>>

tipo::= string | integer

Exemplo: Incluir_predicado ('Is_a',{subclasse**:** string; superclasse**:** string })

Este comando inclui o predicado 'Is_a' cujos parâmetros são subclasse e superclasse, ambos do tipo string.

4.2.4 - REGRAS ATIVAS

As regras no GeRATOM são definidas como instâncias de meta-classes especiais do banco de dados, sendo assim tratadas como qualquer outro objeto do banco de dados. As classes são organizadas segundo o esquema da figura 4.5.

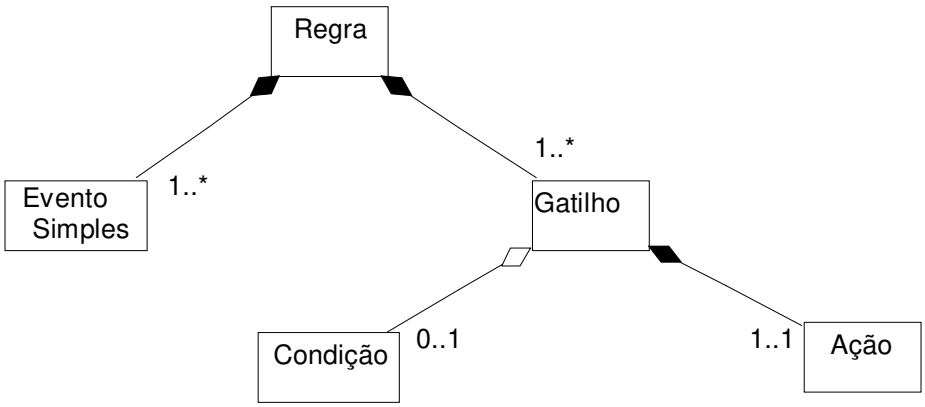

Figura 4.5 Classes básicas do GeRATOM

A definição das regras no GeRATOM foi herdada do TOM-Rules[4, 25].

Evento

No GeRATOM, eventos são gerados através da execução das operações elementares do TOM, que correspondem à criação, exclusão ou alteração de alguma propriedade de um objeto do BD. A definição de um evento obedece a seguinte sintaxe:

Evento::= Evento_Simples

Evento_Simples::= create | delete | establish | remove | update | gr_insert | gr_delete

Exemplo:

Create (objeto1,classe)

Establish (objeto1, relacionamento,objeto2)

O sistema suporta uma forma de evento composto, disjunção de evento (E1 OR E2), que é caracterizada quando uma regra é associada a mais de um evento simples. Portanto, temos como eventos:

Evento::= Evento_Simples | Evento_Composto

Evento_Composto::= Simples OR Simples {OR Simples}

Os eventos simples são definidos através da classe 'Evento' (Figura 4.6)

Class EVENTO
  Class methods
    Create_Event
    Remove
  Instance relationship
    Of-rules: REGRA(0,*)
    Name: STRING(1,1)
    Active: {Y,N}(1,1)
  Instance Methods
    Able
    Disable
    Signal

Figura 4.6 Classe Evento

- **Propriedades**

♦ **Name** – Define uma identificação para uma instância da classe;

♦ **Of-rules** - Determina quais regras são disparadas em conseqüência da ocorrência do evento;

♦ **Active** - Determina se a ocorrência do evento deve ou não ser monitorada pelo sistema.

- **Métodos**

♦ **Create_Event** – Método para cria um novo evento.

Sintaxe: Create_event (<nome_evento>, <status>)

<status>::= 0 | 1

onde 0 = evento desativo e 1 evento ativo

Exemplo: Create_Event (‘Evento1’, ‘’,1)

- **Remove** – Método para excluir um evento existente no sistema.

  Sintaxe : Remove (<nome_evento>)

  Exemplo: Remove (‘Evento1’)

- **Able** - Método para habilitar o monitoramento do evento pelo sistema

  Sintaxe: Able (<nome_evento>).

  Exemplo: Able <Evento1>

- **Disable** - Método para desabilitar o monitoramento do evento

  Sintaxe: Disable(<nome_evento>)

  Exemplo: Disable(Evento1)

- **Signal** - Método para sinalizar um evento e enviar mensagens para disparar as regras associadas.

  Sintaxe: Signal (<operação>, <parâmetros>)

  onde <parâmetros>::= parâmetros de <operação>

  Exemplo: Signal (‘Create’,{objeto1,classe})

Atualmente, o GeRATOM não suporta a definição de eventos temporais. A especificação de eventos temporais poderá ser incluída no sistema, sem a necessidade de modificar a estrutura das classes básicas, pois poderão ser definidos como especializações da classe ‘Evento’, herdando assim, toda a interface definida na classe para a sinalização de um evento.

Podemos verificar que a manipulação de eventos no GeRATOM é realizada através dos métodos da própria classe Evento. Então, para a manipulação de eventos, o sistema permite: incluir, excluir e mudar o status do evento (Figura 4.7).

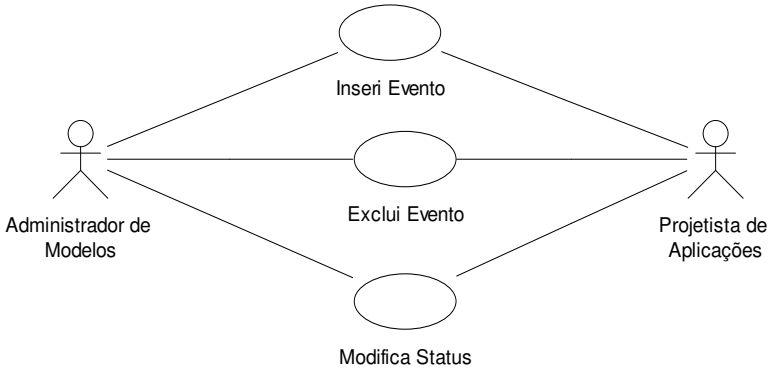

Figura 4.7 Manipulação de Eventos

Condição

A condição determina um estado que deve ser satisfeito para que a ação do gatilho seja executada. Uma condição é definida através da classe "Condição".

```
Class CONDIÇÃO
Class Methods
      Create_condition
      Remove
Instance Relationship
     Expressão: STRING(1,1)
     Alias: STRING(1,1)
Instance Methods
     Eval_condition
```

Figura 4.8 Classe Condição

- **Propriedades**

- **Expressão** – Corresponde à expressão que será avaliada para determinar se a ação de um gatilho será ou não executada.
- **Alias** – Determina quais as variáveis da condição que receberão os parâmetros passados pelo evento antes do início da avaliação da condição. As variáveis referenciadas nesta propriedade devem existir na expressão da condição.

- Métodos

- **Create_condition** – Método que cria uma instância da classe "Condição" correspondendo à condição de um gatilho.

  Sintaxe: Create_condition (<expressão_condição>, <variaveis_alias>)

  Exemplo: Create_condition(**'is_a(x,C) and not x in D'** , **'x,C'**)

- **Remove** – Método que elimina uma instância da classe ‘Condição.

  Sintaxe: <Condição>.Remove

  Exemplo: Condição1.Remove

- **Eval_Condition** – Método utilizado para avaliar a expressão da condição

  Sintaxe: <Condição>.Eval_Condition(<parâmetros>)

  onde parâmetros ::= parâmetros passados pelo evento

A condição de um gatilho é uma expressão formada por predicados estruturais predefinidos ou definidos pelo usuário e predicados comparativos sobre as propriedades dos objetos do BD e é definida obedecendo a seguinte sintaxe:

Condição ::= Predicado {OPERADOR Predicado} | vazio
OPERADOR::= AND | OR
Predicado ::= Estruturais | Comparativos | NOT(Predicado)
Estruturais ::= Predefinidos | Do_Usuário
Comparativos ::= Propriedade_Objeto OPCOMP Propriedade_Objeto
| Propriedade_Objeto OPCOMP Constante
Propriedade_Objeto::= <objeto>.<propriedade>
OPCOMP ::= > | < | = | <> | <= | >=

Ação

A ação de um gatilho determina as operações que serão realizadas de acordo com o resultado da avaliação da condição do gatilho. Sua definição é feita através da classe AÇÃO mostrada a seguir.

Class AÇÃO
Class Methods
Create_action
Remove
Instance Relationships
Action:STRING(1,1)
F_action:STRING(1,1)
Instance Methods
Execute

Figura 4.9 Classe Ação

- **Propriedades**

♦ **Action** – Operação que será executa caso a condição seja avaliada como verdadeira;

♦ **F-action** – Operação executada caso a condição seja falsa.

A especificação da ação e da f-ação de um gatilho deve obedecer a seguinte sintaxe:

Ação ::= create | delete | establish | remove
| update | gr_inser | gr_insert | gr_delete | Abort | Op_usuário

F-ação::= create | delete | establish | remove
| update | gr_insert | gr_delete

- **Métodos**

♦ **Create_action** – Método que cria uma instância da classe 'Ação' correspondendo à ação e f-ação de um gatilho

Sintaxe: Create_action(<ação>,<f-ação>)

Exemplo: Create_action (**'create(x,D)'** , 'Abort')

♦ **Remove** – Método que exclui uma instância da classe 'Ação'

Sintaxe: <Ação>.Remove

Exemplo: Ação1.Remove

♦ **Execute** – Método que executa a ação ou f-ação de acordo com o resultado a avaliação da condição do gatilho.

Sintaxe: <Ação>.Execute (<resultado_condição>)

<resultado_condição> ::= TRUE | FALSE

Exemplo: Ação1.Execute (TRUE)

Gatilhos

Os gatilhos são formados por:

- uma condição, que é opcional. Caso não seja especificada uma condição para um gatilho, sua ação é executada imediatamente após a ocorrência do evento;

- uma ação e uma f-ação. A especificação da f-ação é opcional.

Um gatilho é definido através da classe 'Gatilho'

```
Class GATILHO
  Class Methods
      Create_trigger
      Remove
  Instance relationship
     Name: STRING(1,1)
     Condition: CONDIÇÃO(1,1)
     Action: AÇÃO(1,1)
     Priority: {0...10}
     Status: {0,1}
     Seq_exec: {Before,Equal}
  Instance Methods
      Fire
      Able
      Disable
      Change_priority
      Change_action
      Change_condition
```

Figura 4.10 Classe Gatilho

- Propriedades

- **Name** – Define um nome para o gatilho.
- **Condition** – Propriedade que determina a condição de disparo da ação do gatilho. Propriedade do tipo CONDIÇÃO (classe básica do sistema).
- **Action** – Propriedade que determina as ações (ação e f-ação) que deverão ser executadas de acordo com o resultado da avaliação da condição. Propriedade do tipo AÇÃO (classe básica do sistema).
- **Priority** – Determina a ordem de execução de um gatilho em relação aos outros gatilhos do sistema.
- **Status** – Indica se um gatilho deve ou não ser disparado pelo sistema.
- **Seq_exec** – Determina se o gatilho deve ser disparado antes (before) ou depois (equal) do acontecimento de um evento.

- **Métodos**

- **Create_trigger** – Método para criar um novo gatilho.

  Sintaxe: Create_trigger (<nome_gatilho>, <condição>, <alias>,

<ação>, <f-ação>, <prioridade>, <status>, <seq_exec>)

onde <prioridade> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10
<status>::= 0 (habilitado) | 1 (desabilitado)
<seq_exec>::= Before | Equal

Exemplo: Create_trigger('Gatilho1', Condição1, 'x,C', Ação,'1', '1' 'Equal')

- **Remove** – Método para excluir um gatilho do sistema
  Sintaxe: Remove (<nome_gatilho>)
  Exemplo: Remove ('Gatilho1')

- **Fire** – Método para disparar um gatilho.
  Sintaxe : <gatilho>.Fire
  Exemplo: Gatilho1.Fire

- **Disable** – Método para desativar um gatilho
  Sintaxe: Disable (<nome_gatilho>)
  Exemplo: Disable (Gatilho1)

- **Able** – Método para ativas um gatilho
  Sintaxe: Able (<nome_gatilho>)
  Exemplo: Able (Gatilho1)

- **Change_Priority** – Método para mudar a prioridade de execução de um gatilho
  Sintaxe: Change_Priority (<nome_gatilho>,<valor_prioridade>)
  Exemplo: Change_priority(Gatilho1, 2)

- **Change_action** – Método para mudar a operação que um gatilho deve executar caso sua condição seja verdadeira.
  Sintaxe: Change_action (<nome_gatilho>, <ação>, <f-ação>)
  Exemplo: Change_action(Gatilho1, Ação1**,** F_ação1)

- **Change_condition** – Método para mudar a condição de execução de uma ação

  Sintaxe: Change_condition(<nome_gatilho>, <condição>, <alias>)

  Exemplo: Change_condition(Gatilho1, 'is_a(x,C) and not x in D' , 'x,C')

Regra

As regras são associações entre eventos e gatilhos. Em uma regra é possível associar vários eventos, caracterizando um evento composto, e vários gatilhos.

Uma regra é definida como instância da classe "Regra".

```
Class REGRA
  Class Methods
    Create_Rule
    Remove
  Instance relationship
    Name: STRING(1,1)
    Events: EVENT(1,*)
    Triggers: GATILHO(1,*)
    Priority: {0..10}
  Instance Methods
    Signal
    Change_priority
    Add_event
    Add_trigger
```

Figura 4.11 Classe Regra

- **Propriedades**

- **Name** – Define um nome para a regra.
- **Events** – Corresponde aos eventos que podem disparar a regra. Se mais de um evento foi incluído, temos a caracterização de um evento composto por disjunção.
- **Triggers** – Corresponde aos gatilhos que serão considerados, caso a regra seja disparada.
- **Priority** – Determina a ordem de execução de uma regra em relação às outras regras do sistema.

- **Método**

- **Create_rule** – Cria uma nova instância da classe 'Regra'

  Sintaxe: Create(<nome_regra>, {<eventos>},{<gatilhos>}

onde <eventos>::= <nome_evento> {, < nome_evento>}
<gatilhos>::= <nome_gatilho> {, <nome_gatilho>}

Exemplo: Create_rule (‘Regra1’, {‘Evento1’}, {‘Gatilho1’})

- **Remove** – Exclui uma instância da classe ‘Regra’

  Sintaxe: Remove (<nome_regra>)

  Exemplo: Remove(‘Regra1’)

- **Signal** – Dispara uma regra e inicia o processamento dos gatilhos associadas

  Sintaxe: <Regra>.Signal(<parâmetros>)

  onde <parâmetros>::= parâmetros passados pelo evento

- **Add_event** - Adiciona um evento ao conjunto de eventos que disparam a regra.

  Sintaxe: Add_event(<nome_regra>, <nome_evento>)

  Exemplo: Add_event(‘Regra1’,‘Evento2’)

- **Add_trigger** – Adiciona um gatilho ao conjunto de gatilhos disparados pela regra.

  Sintaxe: Add_trigger(<nome_regra>, <nome_gatilho>)

  Exemplo: Add_trigger(‘Regra1’,‘Gatilho1’)

- **Change_Priority** – Método para mudar a prioridade de execução de uma regra.

  Sintaxe: Change_Priority (<nome_regra>,<valor_prioridade>)

  Exemplo: Change_priority(‘Regra1’, 2)

FLUXO DE MENSAGEM DURANTE O PROCESSAMENTO DE UMA REGRA

Definidas as classes básicas do GeRATOM, mostraremos a seguir, através de um diagrama de colaboração (Figura 4.12), a troca de mensagens entre as classes básicas durante o processamento de uma regra.

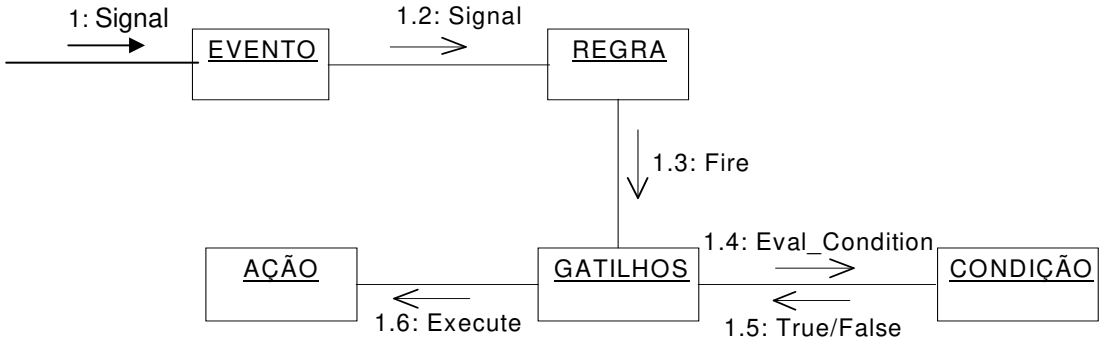

Figura 4.12 Colaboração entre as classes básicas no processamento de um regra.

Uma mensagem *"Signal"* é enviada ao evento (instância da classe Evento) detectado. Ao receber a sinalização, o evento verifica quais regras (instâncias de Regra) são por ele disparadas (propriedade *Of-rules*) e envia a mensagem *"Signal"* para cada regra. Ao receber a mensagem, as regras verificam quais são seus gatilhos (instâncias de Gatilho), disparando-os em seguida, através do envio da mensagem "*Fire*" para cada um.

Cada gatilho, então, inicia o processamento de sua condição (instância de Condição) enviando a mensagem "*Eval_Condition*" e espera o resultado da avaliação. Ao receber o resultado da avaliação, o gatilho envia uma mensagem "*Execute"* a sua ação (instância de Ação), passando o resultado da avaliação da condição. Baseado no resultado, a instância de classe "Ação" executa a ação ou a f_ação do gatilho.

4.3 - SEMÂNTICA DE EXECUÇÃO DE REGRAS

Entende-se por semântica de execução de regras a maneira como regras ativas são processadas e como o processamento dessas regras deve interagir com o BD.

4.3.1 - MODOS DE ACOPLAMENTO

Modos de acoplamento determinam os relacionamentos entre a ocorrência de um evento, a avaliação da condição de uma regra e o processamento da ação.

Para garantir o controle de integridade no TOM, o GeRATOM considera diferentes modos de acoplamento para os pares evento-condição e condição-ação dos ADs e ECs.

Axiomas Dinâmicos (AD)

Um axioma dinâmico é uma regra que estabelece uma condição para que uma determinada operação possa ser realizada no BD. Se a condição for violada, a operação é cancelada através da execução da operação 'Abort'.

Então, para garantir que os ADs tenham o efeito desejado sobre a execução da operação, é necessário que o par Evento-Condição tenha o acoplamento de execução imediato, ou seja, ao ocorrer o evento, a condição deve ser avaliada logo em seguida para verificar se a operação violará ou não alguma restrição.

A consideração do par Condição-Ação também tem que ser imediata, pois caso uma restrição seja violada, o cancelamento da operação deve ser executado imediatamente após a constatação da violação. Portanto, temos também, o acoplamento imediato para o par Condição-Ação.

Efeitos Colaterais (EC)

Os efeitos colaterais são regras que executam operações complementares às operações elementares do TOM. As operações elementares possuem vários ECs associados que verificam a integridade do BD depois da execução da operação elementar e executam operações corretivas sobre o BD.

As condições dos efeitos colaterais verificam o estado do BD resultante da realização de uma operação elementar. Caso alguma violação estrutural seja detectada, o EC executa sua ação para recompor o BD a um estado consistente.

Como várias ECs (regras do modelo) são disparados após a execução de uma operação é necessário garantir que a execução de uma operação corretiva de um EC não interfira na verificação da condição de outro EC.

Para evitar a interferência entre os ECs, determinamos que a execução da ação de um EC é adiada em relação à avaliação de sua condição até que todos os ECs sejam considerados.

Para o par Evento-Condição, determinamos que a avaliação da condição é iniciada imediatamente após a realização de uma operação elementar.

Os modos de acoplamentos para os pares Evento-Condição e Condição-Ação dos ADs e ECs são resumidos na seguinte tabela.

	Evento-Condição	**Condição-Ação**
Axiomas Dinâmicos	Imediato	Imediato
Efeitos Colaterais	Imediato	Postergado

Tabela 4.4 Modos de Acoplamento EC/CA no GeRATOM

4.3.2 - PRIORIDADE

O GeRATOM suporta o esquema de prioridades para resolver possíveis casos de conflitos durante a execução das regras.

Estabelecemos dois esquema de prioridades, um entre as regras e outro entre os gatilhos de um regra, pois, é possível associar mais de um gatilho a uma mesma regra.

Com as prioridades, podemos estabelecer uma ordem de execução entre um conjunto de regras ou entre um conjunto de gatilhos dentro de uma regra. As propriedades 'prioridade' das classes Regra e Gatilho podem assumir valores de 0 a 10, onde, quando maior o valor maior a prioridade de execução.

Por exemplo, sejam dois gatilhos G1 e G2 de uma regra R1 criados com prioridades 4 e 6, respectivamente. Quando a regra R1 tiver sua condição avaliada como verdadeira, o gatilho G2 será disparado antes do gatilho G1, pois sua prioridade de execução é maior que a de G1.

4.3.3 - TERMINAÇÃO

A execução da ação de um EC é uma operação elementar, portanto, é possível que sua execução dispare regras que já foram disparadas anteriormente, gerando eventos repetidos e reiniciando o processamento de regras que já foram processadas.

O problema de terminação é tratado em maiores detalhes na seção 5.7 do próximo capítulo, mas em rápidas palavras, consiste em verificar, antes da execução de uma operação (ação de EC), se a operação pertence a um conjunto de operações já

realizadas. Se a operação pertencer ao conjunto, uma nova execução da operação provocará a execução de outras operações já realizadas, gerando eventos repetidos. Então, para evitar este problema, a nova execução da operação é cancelada.

4.3.4 - DETECÇÃO DE EVENTOS

No GeRATOM, a geração de um evento relativo à execução de um operação elementar é feita de duas formas:

- Externa ao GeRATOM. Acontece toda vez que uma operação elementar é realizada pelo usuário e corresponde a sinalização da operação original;
- Interna ao GeRATOM. Acontece quando a ação de uma regra vai ser realizada.

4.3.5 - DISPARO DE REGRAS CONSIDERANDO O FATOR CLASSE

Muitas regras podem ser associadas a um mesmo evento, causando o disparo simultâneo de várias regras em virtude do acontecimento do evento.

No TOM, muitos ADs e ECs são associados à execução de uma mesma operação elementar, consequentemente, teremos muitas regras do modelo disparadas em decorrência do acontecimento de uma única operação.

Se levarmos em consideração apenas o evento para disparar as regras do modelo, muitas regras desnecessárias serão disparadas, prejudicando o desempenho do sistema. Por exemplo, considerando que

O evento EV1 tem associado às regras R1, R2, R3, R5, R8 e R10;
O evento EV2 tem associado às regras R4, R6, R7, R9 e R11;
A classe C1 é associada às regras R1, R2 e R4;
A classe C2 com as regra R3, R4 e R5.

e que em determinado momento, ocorre a execução de uma operação OP1(C1) modificando a classe C1. Supondo que a operação OP1 sinalize o evento EV1, teremos

então as regras R1, R2, R3, R5, R8 e R10 disparadas, ou seja, as regras R3, R5, R8 e R10 que não têm relação alguma com C1 foram consideradas.

Para evitar este problema, as regras são associadas diretamente às classes das aplicação, além da associação com os eventos. Dessa forma, incluímos o fator "classe" como mais um critério de seleção das regras que serão disparadas, ou seja, a partir de então, o disparo das regras depende do evento e da classe alvo da operação que sinalizou o evento.

Como resultado, teremos que as regras disparadas quando um evento é gerado, corresponderá ao conjunto interseção das regras associadas ao evento e das regras associadas à classe alvo. Então, para o nosso exemplo

$$\text{Regras(EV1)} \cap \text{Regras(C1)} = \{R1, R2, R3, R5, R8, R10\} \cap \{ R1, R2, R4 \} = \{R1, R2\}$$

indicando que apenas as regras R1 e R2 serão processadas para aquela operação.

Com essa operação diminuímos o número de regras disparadas em decorrência de um evento, consequentemente o impacto do processamento de regras desnecessárias sobre o sistema.

4.3.6 - PREDICADOS ESTRUTURAIS, PASSAGEM DE PARÂMETROS E A AVALIAÇÃO DA CONDIÇÃO

Descrevemos na seção 3.5 do capítulo anterior, que um dos problemas em processar regras do modelo no TOM-Rules era a falta de um mecanismo para a troca de parâmetros entre a condição e a ação da regra e entre os predicados estruturais que formam a condição.

A passagem de parâmetros entre os predicados é necessária, pois, alguns predicados retornam valores utilizados para o processamento dos próximos predicados da condição ou para a execução da ação da regra.

Então, antes de explicarmos como é feita a avaliação de uma condição e a passagem de parâmetros no GeRATOM, descrevemos os predicados estruturais predefinidos do modelo e quais os resultados que podem ser retornados por cada predicado durante a avaliação da condição de uma regra.

Predicados Estruturais

Predicados estruturais são utilizados na especificação de ADs e ECs e são executados durante a avaliação da condição de uma regra do modelo.

O TOM dispõe de um conjunto de predicados estruturais predefinidos que retornam verdadeiro ou falso, de acordo com um estado dos dados ou alguma característica de definição das classes do BD.

Além de valores lógicos, alguns predicados podem funcionar como consultas predefinidas, bastando para isso, que um dos parâmetros necessários a sua execução seja suprimido, o que indica que seu valor deve ser recuperado.

A seguir, descrevemos cada predicado estrutural predefinido do TOM, mostrando os parâmetros necessário a sua execução e o seu comportamento caso um de seus parâmetros não seja fornecido. A ausência da descrição do comportamento de um predicado na falta de um parâmetro indica que o predicado não funciona como uma consulta predefinida para esta situação.

A) Predicados Primitivos

IN(objeto, classe)

Verificar a existência de um objeto em uma classe, retornado verdadeiro caso o objeto pertença a classe especificada.

IS_REL(objeto1, rel, objeto2)

Verificar se objetos (objeto1 e objeto2) estão associados pelo relacionamento (rel).

CLASS_REL(rel, classe1, classe2)

Verificar se as classes (classe1 e classe2) estão envolvidas em um determinado relacionamento (rel).

IN_REL(classe, rel)

Verifica se uma classe (classe) esta envolvida em um determinado relacionamento (rel).

Se for fornecida apenas a classe, **IN_rel(classe, ?)**, o predicado funcionará como uma consulta pesquisando todos os relacionamento em que a classe esta envolvida, retornando os relacionamentos encontrados. Se a classe não estiver envolvida em nenhum relacionamento, o predicado retornará falso.

MIN(objeto, rel) e MAX(objeto, rel)

Verifica se o número de instâncias, relacionadas a um objeto por um determinado relacionamento, atingiu os limites estabelecidos pelas cardinalidades máxima e mínima do relacionamento. Ambos os predicados, recebe como parâmetros: um objeto e um nome do relacionamento.

INV(rel)

Funciona como uma consulta recuperando o relacionamento inverso de um relacionamento (rel).

B) Generalização

IS_A(subclasse, superclasse)

Verifica se uma classe (subclasse) é subclasse de outra classe (superclasse).

Se apenas o parâmetro "superclasse" for fornecido, **Is_a(?, superclasse)**, o predicado funcionará como uma consulta pesquisando as classe que são suas subclasses, retornando as subclasses encontradas. Caso a classe fornecida não seja uma superclasse de uma generalização, o predicado retornará falso.

Por outro lado, se apenas o parâmetro "subclasse" for fornecido, **Is_a (subclasse,?)** o que será pesquisada é a superclasse da classe fornecida em uma generalização. Caso a classe fornecida não seja uma subclasse o predicado retornará falso.

PAPEL(objeto, superclasse, subclasse)

Em uma generalização pode-se especificar, no momento da sua criação, um critério de pertinência que determina quais instâncias da superclasse podem ser incluídas em uma determinada subclasse.

O critério pode ser:

- **Explícito**. Nenhuma condição é especificada;
- **Implícito.** Uma condição é especificada como uma regra que deve ser satisfeita para que um objeto pertença a determinada subclasse.

Então, o predicado "Papel" testa o critério de pertinência verificando se ele é explícito ou implícito. Se o critério for explícito, o próprio usuário determinará em que subclasse a instância deverá ser inserida. Por outro lado, se o critério for implícito, é verificado se a instância obedece a sua condição de pertinência de alguma subclasse da generalização. Se a condição for obedecida, o predicado retornará verdadeiro e o nome da subclasse, caso contrário, retorna falso.

DISJUNTIVO(superclasse, subclasses)

Verifica se uma classe (superclasse) é uma generalização disjunta de outras classes (subclasses).

Se apenas o parâmetro "superclasse" for fornecida, **Disjuntivo(superclasse, ?)**, o predicado funcionará como uma consulta pesquisando as subclasse da superclasse em uma generalização disjunta. Se a classe fornecida não for uma superclasse ou a generalização não for disjunta, o predicado retornará falso.

Por outro lado, se apenas o parâmetro "subclasse" for fornecida, **Disjuntivo(?, subclasse)**, o predicado funcionará como uma consulta pesquisando a superclasse da classe fornecida em uma generalização disjunta. Se a classe fornecida não for uma subclasse ou a generalização não for disjunta, o predicado retornará falso.

COVERING(superclasse, subclasses)

Verifica se uma classe (superclasse) é uma generalização completa de outras classes (subclasses).

Se apenas o parâmetro “superclasse” for fornecida, **Covering(superclasse, ?)**,o predicado funcionará como uma consulta pesquisando as subclasse da superclasse em uma generalização completa. Se a classe fornecida não for uma superclasse ou a generalização não for completa, o predicado retornará falso.

Por outro lado, se apenas o parâmetro “subclasse” for fornecida, **Covering(?, subclasse)**, o predicado funcionará como uma consulta pesquisando a superclasse da classe fornecida em uma generalização completa. Se a classe fornecida não for uma subclasse ou a generalização não for completa, o predicado retornará falso.

IN_SUBCLASS(objeto, subclasse_X, superclasse)

Este predicado é utilizado no controle da integridade de generalizações completas ou disjuntas e verifica se um objeto (objeto) pertence a uma subclasse da “superclasse” que seja diferente de “subclasse_X”.

O predicado funciona da seguinte forma: a partir de “superclasse” são recuperadas todas as suas subclasses, em seguida para cada subclasse, diferente de “subclasse_X”, é verificado se o objeto pertence à classe. Se o objeto pertencer a alguma subclasse, o predicado retornará o nome da classe ou das classes, caso contrário retornará falso.

Se o parâmetro subclasse_X não for fornecido, **In_subclass(objeto, ?, superclasse)**, o predicado pesquisará o objeto em todas as subclasse de ‘superclasse’, retornando verdadeiro e a subclasse em que o objeto esta incluído, caso contrário, retornará falso.

C)Agregação

IS_PART(objeto_componente, objeto_agregado)

Verifica se um objeto é parte de outro objeto.

Se apenas o parâmetro “objeto_componente” for fornecido, **Is_part(objeto_componente, ?)**, é verificado se existe no BD, um objeto agregado que o objeto_componente é parte. Se existir, o predicado retornará o objeto agregado como resultado, caso contrário, o predicado retornará falso.

Por outro lado, se for fornecido apenas o parâmetro "objeto_agregado", **Is_part(?, objeto_agregado)**, o predicado pesquisará no BD pelo objeto_agregado e retornará seus objetos componentes. Se o objeto fornecido não for uma agregação de outros objetos, o predicado retornará falso.

CLASS_IS_PART(classe_componente, classe_agregada)

Verifica se uma classe é parte de outra classe em uma agregação.

Se for fornecido apenas o parâmetro "classe_componente", **Class_is_part(classe_componente, ?)**, é verificado se a classe participa de uma agregação como parte de outra classe, ou seja, como uma classe componente. Se a verificação for verdadeira, o predicado retornará a classe agregada, caso contrário, retornará falso.

AGREGADO(classe_agregada, classes_componentes)

Verifica se a classe (classe_agregada) é uma agregação das outras classes (classes_componentes).

Se apenas o parâmetro "classe_agregada" for fornecido, **Agregado(classe_agregada, ?)**, o predicado retornará as suas classes componentes, caso classe_agregada seja definida como uma agregação.

AGREGADO_R(classe_agregada, classe1, classe2, rel)

Verifica se uma classe (classe_agregada) é uma agregação de duas outras classes (classe1 e classe2) através de um relacionamento (rel).

Se apenas o parâmetro "rel" for fornecido, **Agregado_r(?, ?, ?, rel)**, é verifica se existe uma agregação por relacionamento, definida a partir do relacionamento (rel). Se existir, o predicado retornará a classe agregada (classe_agregada) e as classes (classe1 e classe2) que participam do relacionamento que define a agregação.

D) Herança

AGG_HERANÇA(classe_agregada, classes_componentes, rel)

Verifica se em uma agregação, as classes componentes herdam, diretamente, alguma propriedade da classe agregada.

Se apenas os parâmetros “classe_agregada” e classes_componentes forem fornecidos, **Agg_Herança(classe_agregada, classes_componentes, ?)**, o predicado funcionará como uma consulta para recuperar a propriedade herdada pelas classes que são partes da classe agregada.

AGG_HERANÇA_COMP(classe_agregada, classes_componentes, rel, função)

Verifica se em uma agregação, a classe agregada tem alguma propriedade (rel) gerada através da computação de propriedades das classes componentes, através de uma determinada função (função).

Se for fornecido apenas a classe agregada e as classes componentes, **Agg_Herança_Comp(classe_agregada, classes_componentes, ?, ?)**, o predicado funcionará como uma consulta para recuperar a propriedade herdada e a operação que deve ser realizada para obter o novo valor da propriedade.

E) Agrupamento

IS-ELEM(objeto_elemento, objeto_grupo)

Verifica se um objeto (objeto_elemento) é elemento de um grupo (objeto_grupo).

Se for fornecido apenas parâmetro “objeto_elemento”, **Is_elem(objeto_elemento,?)**, o predicado verifica se o objeto pertence a algum grupo. Se a verificação tiver sucesso, o predicado retornará o grupo que contém o objeto.

Mas, se for fornecido apenas o parâmetro “objeto_grupo”, **Is_elem(?, objeto_grupo)**, o predicado verificará se o objeto fornecido é um grupo, ou seja, instância de uma classe que define um agrupamento. Se a verificação for verdadeira, os elementos que formam o grupo são recuperados. Caso o objeto fornecido (objeto_grupo) não seja um grupo, o predicado retornará falso.

IN-GROUP(objeto_grupo, classe_grupo)

Verifica se um objeto (objeto_grupo) é instância de uma classe (classe_grupo) que define um agrupamento.

Se for fornecido apenas o parâmetro “classe_grupo”, **In_group(?, classe_grupo)**, o predicado verificará se a classe define um agrupamento. Se a verificação for verdadeira, o predicado retornará as instância da classe, ou seja, os grupos.

Mas, se for fornecido apenas o parâmetro “objeto_grupo”, **In_group(objeto_grupo,?)**, o predicado verificará se o objeto é instância de alguma classe que define um agrupamento. Se a verificação for verdadeira, o predicado retornará a classe que define o agrupamento, ou seja, a classe grupo.

IS-GROUP(classe_grupo,classe_elemento)

Verifica se uma classe A (classe_grupo) é um agrupamento de uma classe B (classe_elemento).

Se apenas o parâmetro “classe_grupo” for fornecido, **Is_group(classe_grupo,?)**, o predicado verificará se a classe fornecida como parâmetro define um agrupamento. Se a verificação for verdadeira, o predicado retornará a classe elemento do agrupamento, caso contrário, retornará falso.

Mas, se apenas o parâmetro “classe_elemento” for fornecido, **Is_group(?, classe_elemento)**, o predicado verificará se existe um agrupamento onde a classe fornecida participa como classe elemento e retornará sua classe grupo.

GR-DISJUNTIVO(classe_grupo, classe_elemento)

Verifica se uma classe (classe_grupo) é um agrupamento disjunto de outra classe (classe_elemento).

Se apenas o parâmetro “classe_grupo” for fornecido, **Gr_disjuntivo(classe_grupo, ?)**, o predicado verificará se existe um agrupamento disjunto onde a classe fornecida é a classe que define o agrupamento. Se existir, o predicado retornará a classe elemento do agrupamento, caso contrário, o predicado retornará falso.

Por outro lado, se for fornecido apenas o parâmetro “classe_elemento”, **Gr_disjuntivo(?, classe_elemento)**, o predicado verificará se existe um agrupamento disjunto onde a classe fornecida é a classe elemento do agrupamento. Se existir, o predicado retornará a classe que define o agrupamento, ou seja, a classe grupo. Caso não exista o agrupamento, o predicado retornará falso.

GR-COVERING(classe_grupo, classe_elemento)

Verifica se uma classe (classe_grupo) é um agrupamento completo de outra classe (classe_elemento).

Se apenas o parâmetro "classe_grupo" for fornecido, **Gr_covering(classe_grupo, ?)**, o predicado verificará se existe um agrupamento completo, onde a classe fornecida é a classe que define o agrupamento. Se existir, o predicado retornará a classe elemento do agrupamento, caso contrário, o predicado retornará falso.

Por outro lado, se for fornecido apenas o parâmetro "classe_elemento", **Gr_covering(?, classe_elemento)**, o predicado verificará se existe um agrupamento completo onde a classe fornecida é a classe elemento do agrupamento. Se existir, o predicado retornará a classe que define o agrupamento, ou seja, a classe grupo. Caso não exista o agrupamento, o predicado retornará falso.

GR_PAPEL(objeto, grupo, classe_grupo)

Em um agrupamento, similar a generalização, pode-se especificar, no momento de sua criação, um critério de pertinência que determina quais instâncias da classe elemento podem ser incluídas em um determinado grupo (instância da classe grupo).

O critério pode ser:

- **Explícito**. Nenhuma condição é especificada;
- **Implícito.** Uma condição é especificada como uma regra que deve ser satisfeita para que um objeto pertença a um grupo.

Então, o predicado "Gr_Papel" testa o critério de pertinência verificando se ele é explícito ou implícito. Se o critério for explícito, o próprio usuário determinará em que grupo a instância deverá ser inserida. Por outro lado, se o critério for implícito, é verificado se a instância obedece a condição de pertinência de algum grupo. Se a condição for atendida, o predicado retornará verdadeiro e o grupo, caso contrário, retornará falso.

ONLY-IN-GROUP(objeto, objeto_grupo)

Verifica se existem objetos que pertencem a apenas um grupo (objeto_grupo).

Se apenas o parâmetro "objeto_grupo" for fornecido, **Only_in_group(?, objeto_grupo)**, o predicado funcionará como uma consulta para recuperar as instâncias da classe elemento que pertencem **apenas** ao grupo (objeto_grupo) passado como parâmetro.

IN-ANOTHER-GROUP(objeto, objeto_grupo1, classe_grupo)

Verifica se um determinado objeto (objeto) pertence a um grupo, diferente de objeto_grupo1. Este predicado recebe como parâmetros: um objeto, um grupo (objeto_grupo) e a classe grupo que contém o grupo fornecido como parâmetro.

F) Tempo

TIMED_CLASS(classe)

Verifica se uma classe é temporal.

TIMED_REL(rel)

Verifica se um relacionamento é temporal.

EXCL_PRE(pré-classe, classe)

Verifica se uma classe tem precedência de inclusão sobre outra classe, ou seja, o objeto incluído de uma classe, obrigatoriamente, tem que ter vindo de outra classe.

Se apenas o parâmetro "classe" for fornecido, **Excl_pre(?, classe)**, o predicado verificará se existe uma classe que precede (pré-classe) a classe fornecida. Se a verificação for verdadeira, o predicado retornará a classe encontrada, ou seja, a pré-classe, caso contrário, retornará falso.

EXCL_POS(classe, pos-classe)

Verifica se uma classe procede outra classe com relação a operação de exclusão, ou seja, um objeto excluído de uma classe, obrigatoriamente, tem que ser incluído em outra classe.

Se apenas o parâmetro "classe" for fornecido, **Excl_pos(classe, ?)**, o predicado verificará se existe uma classe que procede (pós-classe) a classe fornecida. Se a verificação for verdadeira, o predicado retornará a classe encontrada, ou seja, a pós-classe, caso contrário, retornará falso.

O(objeto, subclasse, superclasse)

Em uma generalização, depois da alteração de um relacionamento é possível que um objeto não obedeça mais o critério de pertinência de uma subclasse, sendo, portanto, necessário removê-lo da classe.

Este predicado verifica o próximo estado de um objeto, ou seja, se existir uma subclasse, diferente de "subclasse", que tem seu critério de pertinência atendido pelo objeto depois que a modificação for realizada. Se existir, o predicado retornará o nome da subclasse, caso contrário retornará falso.

OVERLAP(objeto, tempo1, tempo2, classe)

Verifica se o objeto que esta sendo inserido em uma classe temporal já existe no BD e é válido naquele momento, ou seja, **<objeto,t1,*now*>**, onde t1 é a data de sua criação no BD e *now* representa o momento presente. Se o objeto existir, é verificado se a data de sua criação no BD é menor que a data de criação do novo objeto **(t1<tempo1)** e **tempo2=*'now'***, retornando verdadeiro se as duas condições forem satisfeitas.

O predicado recebe como parâmetros: um objeto, o intervalo de validade do objeto no BD e a classe.

Descritos os predicados estruturais e os resultados por eles retornados, mostramos a seguir, como os resultados são utilizados durante o processamento de uma regra.

Avaliação da Condição e a Passagem de Parâmetros

Para permitir o processamento adequado das regras do modelo, o GeRATOM disponibiliza um mecanismo para a passagem de parâmetros entre as partes de uma regra e entre os predicados que formam a sua condição.

A passagem de parâmetros entre o evento e a condição da regra, é feita através do método que gerou o evento e da propriedade “Alias” da classe Condição.

Quando uma operação é realizada, seus parâmetros são passados juntos com a sinalização do evento que representa a execução da operação. Ao receber a sinalização, o Monitor de eventos, subsistema do GeRATOM, avisa ao sistema da ocorrência do evento, iniciando o processamento das regras associadas ao evento e, consequentemente, o processamento de seus respectivos gatilhos.

Antes de avaliar a condição de cada gatilho, é criada a lista de variáveis de transição da condição, que é composta pelas variáveis que forma a condição do gatilho. Criada a lista e baseado nos valores da propriedade “Alias” da condição, é feita a associação entre os parâmetros passados pelo evento e as variáveis da condição do gatilho, caracterizando assim, a passagem de parâmetros entre evento e condição.

Depois que a lista de transição é criada e os valores passados pelo evento são associados às variáveis, é iniciada a avaliação da condição.

Toda vez que um predicado da condição vai ser avaliado, a lista de transição é consultada para verificar quais os valores associados às variáveis passadas com parâmetros para o predicado. Finalizado o processamento do predicado, é feita a atualização dos valores na lista, de acordo com os dados retornados pelo predicado avaliado.

A atualização da lista depois da avaliação de um predicado é necessária, pois, em alguns casos, os predicados funcionam como consultas predefinidas retornando valores que podem ser utilizados pelos próximos predicados da condição ou para executar a ação do gatilho.

Finalizada a avaliação da condição, as variáveis da lista de transição têm a elas associadas, os valores resultantes do processamento dos predicados que forma a condição do gatilho. Então, através de uma nova consulta na lista, são obtidos os valores das variáveis passadas como parâmetros para ação da regra, caracterizando a passagem de parâmetros entre a condição e a ação.

Para o melhor entendimento de como é feita a passagem de parâmetros no GeRATOM, mostramos a seguir, o processamento de uma regra do modelo desde a ocorrência do evento até a execução da ação da regra.

Processamento de uma Regra

Para mostrar como é feito o processamento de uma regra do modelo utilizaremos a seguinte regra.

ON Create (x,C)
IF is_a(C,D) and not x in D *{Alias = x,C}*
THEN Create(x,D)

Esta regra é um efeito colateral do conceito "Generalização" e verifica se um objeto que foi inserido em uma subclasse existe na superclasse. Caso não exista, o objeto é criado pela ação da regra. Os alias desta regra são "x" e "C".

Considerando a ocorrência da operação 'Create(10,'carro_em_uso')'. O evento 'Create' é sinalizado e os seguintes parâmetros são passados: 10 e 'carro_em_uso'.

A regra é sinalizada e o gatilho associado é disparado, no caso,

IF is_a(C,D) and not x in D *{Alias = x,C}*
THEN Create(x,D)

Antes do inicio da avaliação da condição do gatilho, é criada a lista das variáveis de transição da condição

Variáveis de Transição	X	C	D

e os parâmetros passados pelo evento, são atribuído as variáveis definidas como *"Alias"*. Para o gatilho do exemplo, temos a lista de variáveis com os seguintes valores

Variáveis de Transição	X	C	D
Valores iniciais	**10**	**Carro_em_uso**	

e a **Condição = Is_a(C,D) and not x in D** para ser avaliada.

Uma vez criada a lista com os valores iniciais, o primeiro predicado estrutural da condição **(Is_a)** é avaliado. Supondo que "Carro_em_uso" é subclasse de "Carro", o predicado retornará verdadeiro e a lista será atualizada com os seguintes valores

	X	C	D
Antes da Avaliação	10	Carro_em_uso	
Is_a (C,D)	**10**	**Carro_em_uso**	
Depois da Avaliação	10	Carro_em_uso	Carro

e a **Condição =TRUE and not x in D,** parcialmente avaliada.

Em seguida, será avaliado o predicado estrutural **(In)** com os valores indicados na lista, ou seja, **"10 in Carro".** Supondo que o objeto 10 não pertence à classe "Carro" teremos,

	X	C	D
Antes da Avaliação	10	Carro_em_uso	Carro
X in D	**10**	**Carro_em_uso**	**Carro**
Depois da Avaliação	**10**	**Carro_em_uso**	**Carro**

e a **condição = TRUE and not FALSE**. O valor de "D" não foi modificado, pois o predicado "In" retornou apenas um valor lógico.

Finalizada a análise dos predicados, a condição é avaliada por completo.

Condição = TRUE and not FALSE

Condição = TRUE

Como a condição foi avaliada como verdadeira, a ação será executada com os seguintes valores: x=10 e D='Carro', ou seja, **Create(10, 'Carro').**

Com o GeRATOM, através do mecanismo de inclusão de novos conceitos do modelo ao sistema e de seus ADs e ECs, garantimos a realização da característica aberto do TOM no seu sistema de controle de integridade.

ADs e ECs (regras do modelo) são utilizados para garantir, depois da execução de uma operação elementar, que os dados do BD, estejam de acordo com as características básicas dos conceitos do modelo.

Cada AD e EC é definido apenas em relação a uma operação elementar, mas durante seu processamento deve considerar os objetos, a classe ou o relacionamento modificado pela operação.

A definição em relação a uma operação é considerada no momento de disparar as regras, por exemplo, se uma operação create(42, carro) for executada, os ADs e ECs da operação 'create' são disparados. Mas, depois que os ADs e ECs são disparados, utilizam os objetos modificados pela operação para processar sua condição e ação, no nosso exemplo o objeto com identificação '42' e a classe 'Carro'.

Com o suporte a regras parametrizadas é possível especificar regras em relação apenas a uma operação elementar e, através dos parâmetros, referenciar os objetos modificados pela operação, para realizar o processamento das regras, atendendo dessa forma, os requisitos para a especificação e processamento de ADs e ECs no GeRATOM.

Em relação as regras de aplicação, elas também serão especificadas de forma genérica, mas, diferente das regras do modelo que são associadas aos conceitos, as regras de aplicação são, explicitamente, associadas a um elemento (classe ou relacionamento) da aplicação, determinado que a regra será disparada em relação à operação e, também, ao elemento modificado.

Por exemplo, para a regra que determina que o salário de um empregado não pode baixar (Figura 4.13), temos a operação 'Update' e o relacionamento 'tem_salário' entre as classes Empregado (classe origem) e Salário (classe relacionada).

```
ON update (x, tem_salário, x.tem_salário, novo_salário)
IF x.tem_salário > novo_salário
THEN  abort
```

Figura 4.13 Regra Salário_não_pode_baixar

Então, para que a regra verifique corretamente a restrição, ela é associada, pelo usuário, à classe Empregado (classe origem do relacionamento). Dessa forma, a regra só

será disparada caso uma operação de modificação (update) tente alterar o relacionamento ‘tem_salário’ de um empregado.

5 – PROJETO DO GeRATOM

O GeRATOM é o gerenciador de regras do TOM, desenvolvido para processar regras ativas com o objetivo de garantir a integridade dos dados das aplicações desenvolvidas no TOM. O sistema foi implementado sobre um SGBD relacional utilizando o ambiente de programação Borland Delphi 4.0.

O GeRATOM esta inserido em um projeto mais abrangente, denominado TOP *(Temporal Object Processing)*. A seguir, mostramos como o GeRATOM se coloca dentro do TOP.

5.1 – PROJETO TOP

O projeto TOP pretende desenvolver um ambiente de projeto e gerenciamento de sistema de informações avançadas, tendo como modelo base o TOM. Seus principais módulos são:

- **POKER *(Petri Net Oriented Knowledge Enginnering Research)*** [26]**.** Método de projeto de sistemas de informação que dá ênfase na modelagem dinâmica (de processos) do que na estrutura dos dados;
- **FADO (Ferramenta de Análise e Desenvolvimento de Modelos de Dados)** [15, 18]**.** Método de projeto de sistemas de informação puramente orientado a objetos. Existe uma ferramenta, denominada CASE/FADO, de suporte computacional ao desenvolvimento de aplicações com a metodologia FADO;
- **MetaTOM (Ambiente de Desenvolvimento de Modelos de Dados).** Ambiente que permite adaptar o modelo de dados do sistema para que nova aplicações possam ser desenvolvidas mais adequadamente;
- **ConTOM (Ambiente Visual de Consultas ao Modelo TOM)** [28]. Interface visual de consulta a banco de dados que permite explorar as estruturas complexas e possíveis características temporais dos objetos e classes.
- **GOTA (Gerenciador de Objetos Temporais e Ativos).** SGBD relacional estendido para processar objetos complexos, temporais, textos não-

estruturados e regras ECA. O GeRATOM é um dos componentes deste módulo.

5.1.1 – ARQUITETURA DO GOTA

No esquema da figura 5.1 mostramos a arquitetura do GOTA.

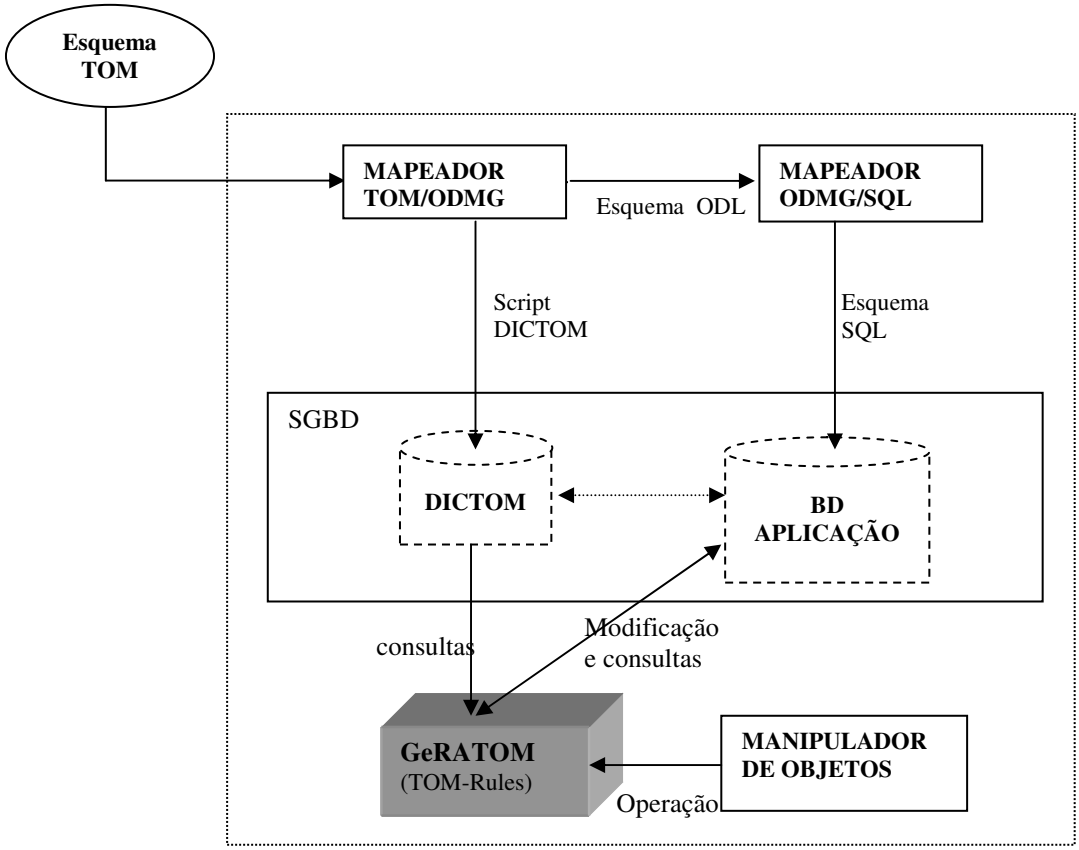

Figura 5.1 Arquitetura do GOTA

Os mapeadores TOM/ODMG e ODMG/OpenBASE[1] [22] são aplicações que combinadas, convertem um esquema conceitual TOM em um esquema OPUS[2]. Os dois mapeadores interagem através de *scripts*, da seguinte forma: o primeiro mapeador recebe um *script* contendo um esquema TOM convertendo-o em um esquema intermediário escrito em ODL/ODMG. O *script* em ODL é passado para o segundo mapeador para ser convertido em um esquema em OPUS.

O DICTOM é o dicionário de dados do modelo e é utilizado diretamente pelo GeRATOM para processar as regras de integridade do modelo.

[1] SGBD da Tecnoocop Sistemas/ RJ.
[2] Linguagem de Definição de Dados suportada pelo OpenBASE

O Manipulador de Objetos (MO) é utilizado para executar as operações elementares do TOM, ou seja, incluir, excluir e modificar objetos do BD. Ele também, antes de executar uma operação elementar, envia uma mensagem ao GeRATOM sinalizando a intenção de executar uma operação, iniciando o processamento dos axiomas dinâmicos da respectiva operação e, dependo do resultado do processamento, a operação será ou não realizada.

Pelo esquema da figura 5.1 podemos verificar que o GeRATOM esta inserido no GOTA e interage diretamente com o Manipulador de Objetos e o DICTOM, então o processamento dos ADs e ECs, especificamente, pelo GeRATOM, depende desta interação, ou seja, das informações passadas ao GeRATOM pelo Manipulador de Objetos e do uso das informações armazenadas no DICTOM e dos próprios dados das aplicações.

Estes recursos não estavam disponíveis no GOTA até o momento, portanto, fez-se necessário, antes de implementar o GeRATOM, a criação do DICTOM e a implementação do Manipulador de Objetos.

Outro aspecto que vamos tratar neste capítulo é a conversão de classes TOM para tabelas realizada para permitir que os dados dos BDs modelados no TOM possam ser armazenados em um BD relacional. Este mapeamento foi considerado para a implementação de alguns predicados estruturais.

5.1.2 – CONVERSÃO DO MODELO TOM PARA O MODELO RELACIONAL

O modelo TOM possui uma linguagem para a definição de dados (LDD) que pode ser utilizada tanto para definir uma classe de um BD, como também, uma metaclasse do modelo. Esta linguagem é mostrada no apêndice A e serviu como base para a geração das informações do DICTOM.

Descrevemos na seção 5.1.1 que o GOTA possui dois mapeadores que são utilizados para converter um esquema de dados TOM em um esquema interno suportado por um SGBD. Este mapeamento é feito em duas etapas: a primeira converte o esquema TOM em um esquema intermediário (OLD/ODMG) e a segunda converte o esquema intermediário em uma representação interna suportada pelo OpenBASE.

Um dos objetivos do GOTA é que ele possa ser agregado a diferentes SGBDs relacionais, enriquecendo assim, seus modelos de dados. Para atingir este objetivo, o segundo mapeador deverá ser reimplementado para gerar um esquema em uma linguagem suportada ao mesmo tempo por diferentes SGBDs relacionais. Então, como os SGBDs relacionais mais populares implementam o padrão SQL/92 [8], este trabalho levou em consideração a sua especificação.

Outro aspecto deste trabalho é que ele não implementa uma nova versão do segundo mapeador, mas, define regras utilizadas para converter classes TOM em uma representação relacional que deverão ser obedecidas em uma possível real reimplementação do segundo mapeador de forma a obter compatibilidade entre os mapeadores e o GeRATOM. As regras de mapeamento são as seguintes:

1) Toda classe estruturada do TOM é convertida em uma tabela com o mesmo nome da classe.

a) A tabela possui um atributo cujo nome é formado pela junção da palavra "codigo_" e do nome da classe e que tem o objetivo de armazenar um identificador atribuído pelo sistema a um objeto para identificá-lo de forma única em relação aos outros objetos do sistema. Este atributo deve ser definido como único;
b) A chave primária da tabela corresponde a chave especificada para a classe;
c) Se a classe for temporal a tabela terá dois atributos adicionais denominados "de" e "ate" para armazenar os intervalos de validade das instâncias da classe temporal no BD.

Exemplo:

Para a classe Carro a seguir será criada a seguinte tabela

class CARRO
instance_relationships
tem_num_reg : NUM_REGISTRO (1,1)
produzido_em : ANO_PRODUCAO (1,1)
tem_num_ser : NUM_SERIE (1,1)
e_do_modelo : MODELO (1,1)
keys_are tem_num_reg, produzido_em, tem_num_ser
generalization_of CAR_DE_FABR, CAR_EM_REVENDA, CAR_EM_USO, CAR_DESTRUIDO **explicit parameters covering, disjunctive parameters_with_time**

Tabela

Carro(Codigo_carro, tem_num_reg, produzido_em, tem_num_ser, de, ate)

- Tem_num_reg, Produzido_em, Tem_num_ser são definidas como a chave primária da tabela
- O relacionamento E_do_modelo não foi convertido em um atributo pois a classe relacionada é uma classe estruturada.
- De e Até foram inseridos pois a classe é temporal (**parameters_with_time**)

2) Toda classe de domínio é absorvida como um atributo de uma tabela.

a) O nome do atributo é o nome do relacionamento que liga a classe de domínio a uma classe estruturada ou não.

Exemplo:

No exemplo da classe Carro as classe TEM_NUM_REG, TEM_NUM_SER E PRODUZIDO_EM foram absorvidas como atributos da tabela "Carro" gerada.

3) Todo relacionamento envolvendo uma classe de domínio e uma classe estruturada e com cardinalidade máxima 1 (um) é transformado em um atributo da tabela que representa a classe estruturada.

a) O nome do atributo é o nome do relacionamento.

Exemplo:

Vide exemplo da classe Carro e das classes TEM_NUM_REG, TEM_NUM_SER e PRODUZIDO_EM.

4) Um relacionamento envolvendo duas classes estruturadas é convertido em um tabela cujo nome corresponde ao nome do relacionamento.

a) A chave primária da tabela é a junção do identificador de objetos da classe origem e o identificador de objetos da classe relacionada pelo relacionamento.

Exemplo:

Na definição da classe Carro o relacionamento **E_do_modelo** que relaciona a classe "Carro" com a classe "Modelo" é convertido em uma tabela com o seguinte esquema

E_do_modelo (Codigo_carro, Codigo_modelo) onde Código_carro e Código_modelo formam a chave primária da tabela.

5) Um relacionamento com cardinalidade máxima maior que 1 (um) é convertido em uma tabela cujo nome é o mesmo do relacionamento.

a) Se o relacionamento envolver duas classes estruturadas os atributos da tabela serão: os identificadores de objetos das classes envolvidas no relacionamento;

Exemplo:

Considerando que a classe MODELO é relacionada com CARRO pelo relacionado E_DO_MODELO que tem como relacionamento inverso o relacionamento E_MODELO_DE que relaciona MODELO com CARRO.

class MODELO
instance_relationships
modelo_tem_nome : NOME (1,1)
tem_cons_comb : CONS_COMBUSTIVEL (1,1)
eh_produzido : FABRICANTE (1,1)
e_modelo_de : CARRO (0,*)
keys_are tem_nome

A tabela gerada no item 4 pela conversão do relacionamento E_DO_MODELO é utilizada como o mapeamento do relacionamento E_MODELO_DE, evitando assim, a geração de duas tabelas para armazenar as mesmas informações.

b) Se o relacionamento envolver uma classe estruturada e uma de domínio os atributos da tabela serão: o identificador de objetos da classe estruturada e a própria classe de domínio.

6) Em uma generalização as tabelas que representam as subclasses possuem um atributo adicional para identificar a que objeto da superclasse uma instância de uma subclasse faz referência. Este atributo é utilizado para armazenar o código

de identificação do superobjeto atribuído pelo sistema e possui o mesmo nome deste atributo na superclasse;

Exemplo:

A classe Carro é uma generalização das classes CAR_DE FABR, CAR_EM_REVENDA, CAR_EM_USO, CARO_DESTRUIDO. Então, considerando que a classe CAR_DE_FABR tem a seguinte definição

class CAR_DE_FABR
class_relationships
post_class : CAR_EM_REVENDA **exclusive**
keys_are inherited
parameters_with_time and life_time 3 year

teremos a geração da tabela Car_de_fabr com o seguinte esquema

Car_de_fabr (Código_car_de_fabr, código_carro, de, ate)

onde Código_car_de_fabr e Código_carro forma a chave primária e código_carro foi herdado da superclasse de Car_de_fabr. Para a geração desta tabela levamos em consideração também a regra 1.

7) Em um agrupamento é criada uma tabela para armazenar os objetos da classe-elemento que pertecem a determinados grupos – objetos da classe-grupo

a) O nome desta nova tabela é a junção de "grupo_" e o nome da classe-grupo;

Exemplo:

Para o agrupamento GRP_PROPRIETARIO definido a seguir e que é utilizado para agrupar as pessoas que foram proprietárias de um determinado carro.

class GRP_PROPRIETARIO
instance_relationships
grupo_tem_nome : NOME(1,1)
keys_are grupo_tem_nome
grouping_of PESSOA **using** E_dono

será gerado a seguinte tabela

Grupo_grp_proprietario (Codigo_grp_proprietario, Codigo_pessoa)

onde a junção dos dois atributos forma a chave primária da tabela.

8) **Em uma agregação a tabela que representa a classe agregada, além de ser formada pelos atributos da classe, possui um atributo correspondendo a cada classe componente da agregação. Estes atributos adicionais armazenarão os códigos de identificação dos objetos que são partes de um objeto agregado.**

Exemplo:

Para a classe Fabr_por_ano definida a seguir

class FABR_POR_ANO
instance_relationships
quant_car_reg : QUANT_CARRO (1,1)
keys_are inherited
aggregation_of ANO_PRODUCAO, FABRICANTE

teremos a geração da seguinte tabela **Fabr_por_ano(Codigo_fabr_por_ano, Ano_Produção, Codigo_fabricante, Quant_carro)**

Observação: O atributo Ano_Produção possui este nome porque foi gerado a partir de uma classe de domínio.

As regras de 1 a 8 foram derivadas, com algumas alterações, das regras de mapeamento descritas em [22].

5.1.3 - DICTOM - DICIONÁRIO DO MODELO TOM

O DICTOM é um dicionário de dados que armazena informações a respeito da definição das classes das aplicações, segundo os conceitos que formam o TOM. Por exemplo, quais classes são temporais, quais os relacionamentos em que determinada classe está envolvida.

Sua criação é necessário pois, como podemos constatar, as regras definidas na seção 5.1.2 são regras que tratam apenas da extensão dos dados não considerando nenhum aspecto intencional o que não é suficiente para manter a integridade dos dados em um BD, pois não temos informações a respeito da definição conceitual das classes.

Então, para garantir a manutenção da integridade dos dados no TOM, ou seja, o processamento correto dos AD's e EC's, definimos um conjunto de tabelas cujo objetivo é armazenar as descrições das classes definidas em um BD modelado no TOM, de forma que possam ser consultadas durante o processamento dos ADs e ECs. A seguir mostramos a definição de cada uma das tabelas do DICTOM.

DICCLASSE (Nome_Bd, Nome_Classe, Domínio, Temporal, Unid_Temp, Lifetime).
DICCHAVE (*[1]Nome_Bd, *Classe, Chave).
DICREL(*Nome_Bd,Nome_Rel,*Classe_Origem,*Classe_Relacionada, Card_Max, Card_Min, Rel_Inverso, Card_Max_Inv, Card_Min_Inv, Valores_Anteriores, Intervalo_Tempo).
DICPOST (*Nome_Bd, *Classe, *Post_Classe, Exclusivo).
DICPRE (*Nome_Bd, *Pre_Classe, *Classe, Exclusivo).
DICGENER(*Nome_Bd, *Classe, *Sub_Classe, Parâmetro, Predicado, Explicito, Rel).
DICPAPEL (*Nome_Bd, *Classe, *Sub_Classe, Nome_Papel).
DICHERANCA_DIRETA(*Nome_Bd,*Classe_Agregada,*Rel_Herdado, *Classe_Q_Herd).
DICHERANCA_COMPUTADA(*Nome_Bd,*Classe_Agregada, *Rel_Computado, *Classe_Origem, *Rel_Origem, Função).
DICAGRUP(*Nome_Bd, *Classe_grupo, *Classe_elemento, Tipo, Parâmetro, Predicado, Rel).
DICAGREG (*Nome_Bd, *Classe_Agregada, *Classe_componente, Tipo, Exclusivo).
DICAGREG_REL(*Nome_Bd,*Classe_Agregada,*Classe_comp1,*Classe_comp2, Rel_de_Agregacao, Exclusivo).
DICCLASSE_TABELA (*Nome_Bd, Nome_Tabela).
DICREL_TABELA (*Nome_Bd, Nome_Tabela).

Cada tabela armazena um tipo de informação útil para o processamento dos ADs e ECs de um conceito do modelo e todas foram definidas considerando a linguagem de definição de dados do TOM e o padrão SQL/92. O padrão SQL/92 foi utilizado para permitir que o DICTOM possa ser criado em qualquer SGBD relacional que suporte o padrão.

[1] Chave Estrangeira

As informações que são armazenadas nas tabelas do DICTOM são extraídas do próprio esquema conceitual da aplicação e são utilizadas para associar as regras semânticas às classes de uma aplicação e durante a avaliação dos predicados estruturais que formam as condições dos ADs e ECs .

GERAÇÃO DAS INFORMAÇÕES DO DICTOM

Os mapeamentos são feitos através de uma leitura linear dos esquemas que serão convertidos, ou seja, a medida que uma determinada palavra é encontrada em um esquema, a palavra equivalente no próximo esquema de conversão é inserida em um novo arquivo que, no final, representará o esquema mapeado em uma nova representação.

Aproveitando-se dessa característica, inserimos no mapeador TOM/ODMG pontos de captação de informações para obter dados sobre as definições das classes de um BD descrito no TOM, definições estas, que serão armazenados nas tabelas do DICTOM. Nenhuma alteração foi feita no mapeador ODMG/OpenBASE, uma vez que este deverá ser reimplementado para um mapeador ODMG/SQL92.

As informações captadas durante o primeiro mapeamento não são inseridas diretamente no DICTOM, mas, são armazenadas temporariamente em várias listas encadeadas até que todo o esquema seja lido pelo mapeador. No final do mapeamento, cada lista é percorrida e, a medida que seus nodos são visitados, são criados comandos de inclusão[1] referentes as informações de cada nodo. Cada comando criado é incluído em um arquivo denominado “DICIONARIO.TOM”.

Após a realização do primeiro mapeamento teremos como resultado dois *scripts:* um contendo o esquema TOM convertido em ODL/ODMG e outro denominado “DICIONARIO.TOM” contendo comandos de inclusão que deverão ser executados para povoar o DICTOM com as definições das classes da aplicação que foi mapeada.

Implementada a aplicação e sua descrição armazenada no DICTOM é possível monitorar violações semânticas através dos ADs e ECs , evitando assim, inconsistências nos dados do BD.

[1] Comandos INSERT de acordo com o padrão SQL/92

5.1.4 - MANIPULADOR DE OBJETOS

O Manipulador de Objetos (MO) (Figura 5.2) é um sistema utilizado para executar as operações elementares do TOM em um ambiente interativo. O sistema também sinaliza a execução das operações ao GeRATOM, passando os parâmetros que são utilizados pelo Monitor para gerar o evento correspondendo a realização da operação.

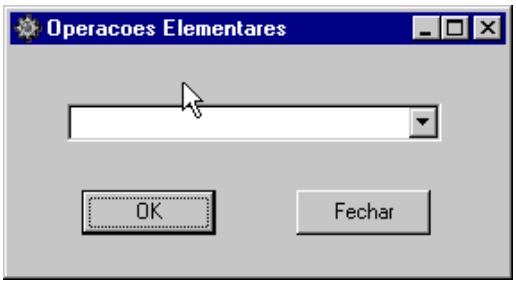

Figura 5.2 Manipulador de Objetos

Com o MO, o usuário pode escolher a operação elementar que deseja executar e fornecer os parâmetros para sua execução.

O MO não executa as operações elementares diretamente no BD, ao invés disso, passa suas intenções de execução ao GeRATOM, e a partir das intenções, o sistema verifica os ADs e ECs das operações e as executa em seguida no BD.

A seguir descrevemos para cada operação: seu objetivo, os parâmetros necessários a sua execução, a maneira como o MO permite sua execução e os parâmetros passados pela operação ao monitor de eventos do GeRATOM.

a) CREATE

Operação utilizada para criar um novo objeto em uma classe, é executada utilizando a janela " Inserir Objeto" (Figura 5.3). O único parâmetro solicitado ao usuário para realizar a operação é o nome da classe que conterá o novo objeto.

Figura 5.3 Janela para criação de um novo objeto.

Durante a criação de um objeto é verificada no DICTOM, se a classe, onde o novo objeto será inserido, está envolvida em relacionamentos com cardinalidade mínima igual a um. Se a verificação for verdadeira e utilizando a janela "Relacionamentos Obrigatórios" (Figura 5.4), é solicitado os objetos que se relacionarão com o novo objeto através dos relacionamentos com cardinalidade mínima um.

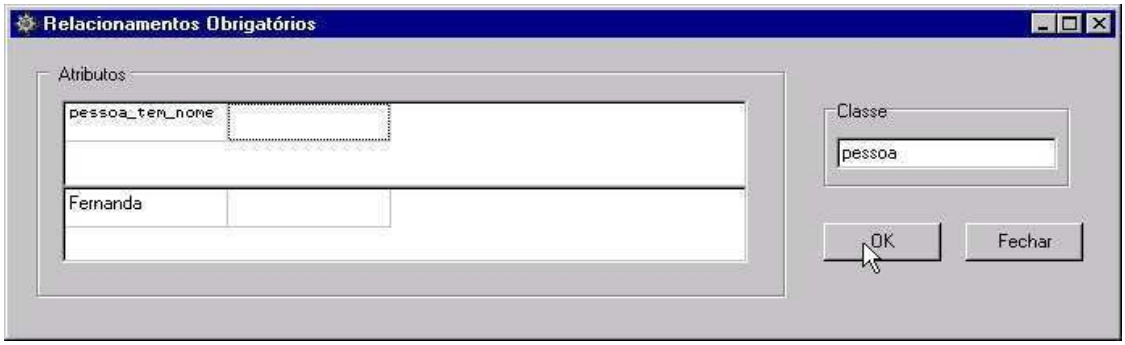

Figura 5.4 Janela com relacionamentos

No momento da criação do objeto, o sistema atribui um identificador único ao objeto (OID) para distingui-lo dos outros objetos do sistema, e passa a intenção de inserir o objeto na classe - **"Create(objetoID, classe)"** ao Monitor.

Os parâmetros passados por esta operação ao "Monitor" são: o nome da operação, o nome da classe modificada e o identificador do objeto inserido.

b) DELETE

Esta operação é utilizada para excluir um objeto de uma classe e necessita dos seguintes parâmetros: o objeto a ser excluído e a classe que contém o objeto, ou seja **"Delete(ObjetoID, classe)".**

No Manipulador de Objetos, a operação é realizada utilizando a janela "Excluir Objeto" (Figura 5.5), que inicialmente, disponibiliza ao usuário, a lista das classes existentes no BD. Com a lista o usuário pode escolher a classe de onde ele deseja

excluir um objeto. Escolhida a classe, é exibida a lista dos objetos que pertencem a classe no momento. A partir da lista de objetos, o usuário escolhe o objeto a ser excluído e em seguida executa a operação.

Figura 5.5 Janela para excluir um objeto.

Através do botão "Detalhes", os relacionamentos em que o objeto está envolvido podem ser consultados antes da exclusão.

Os parâmetros passados por esta operação ao "monitor" são: o nome da operação, o nome da classe modificada e o identificador do objeto excluído.

c) ESTABLISH

Esta operação é utilizada para estabelecer um relacionamento entre instâncias de duas classes e recebe como parâmetros: o nome do relacionamento e os objetos envolvidos no novo relacionamento, ou seja **"Establish(ObjetoID1, rel, ObjetoID2)"**.

No Manipulador de Objetos esta operação é realizada através da janela "Estabelecer Relacionamento" (Figura 5.6), que inicialmente, disponibiliza a lista de todos os relacionamentos existentes no BD. Através dessa lista, o usuário escolher o relacionamento (rel) que deseja estabelecer. Escolhido o relacionamento, são exibidas as classes envolvidas no relacionamento juntamente com as listas de seus respectivos objetos. Nas listas de objetos, o usuário escolhe, os objetos (Objeto1, Objeto2) que estarão envolvidos no relacionamento que será estabelecido.

Figura 5.6 Janela para estabelecer um relacionamento.

Os parâmetros passados por esta operação ao "monitor" são: o nome da operação, o nome do relacionamento e os identificadores dos objetos envolvidos no novo relacionamento.

d) REMOVE

Esta operação é utilizada para excluir um relacionamento existente entre instâncias de duas classes. Para a realização da operação é solicitado o nome do relacionamento e os objetos que terão o relacionamento, entre eles, removido.

Os parâmetros necessário são: nome do relacionamento e os objetos envolvidos no relacionamento que será excluído, ou seja **"Remove (rel, ObjetoID1, ObjetoID2)"**.

Figura 5.7 Janela para remover um relacionamento.

A operação é realizada através da janela "Remover Relacionamento" (Figura 5.7), que inicialmente, disponibiliza a lista de todos os relacionamentos existentes no BD. Através dessa lista, o usuário escolher o relacionamento (rel) que deseja remover. Escolhido o relacionamento, são exibidas as classes envolvidas, acompanhadas com as listas de seus respectivos objetos. Nas listas de objetos, o usuário escolhe, os objetos (Objeto1 e Objeto2) que terão o relacionamento entre eles removido.

Os parâmetros passados ao "monitor" são: o nome da operação, o nome do relacionamento e os identificadores dos objetos afetados pela operação.

e) UPDATE

Em um relacionamento temos dois tipos de classe: a classe origem, classe dominante do relacionamento, e a classe relacionada, classe que é associada a classe origem pelo relacionamento. Por exemplo, no relacionamento "Carro **tem** Dono", temos o relacionamento "Tem" e as classes envolvidas "Carro" e "Dono", onde a classe "Carro" é a classe origem e "Dono" é a classe relacionada.

A operação "Update" é utilizada para alterar um relacionamento existente entre instâncias de duas classes, a classe origem e a classe relacionada.

Para sua execução são necessários: o nome do relacionamento, os objetos envolvidos no relacionamento atual, objeto origem (objeto da classe origem do relacionamento), e o objeto relacionado (objeto da classe relacionada). Outro parâmetro necessário é o objeto que se relacionará ao objeto da classe origem depois do relacionamento ser modificado. Assim, temos **"Update(ObjetoID1, rel, ObjetoID2, ObjetoID3)"**.

A operação é realizada através da janela "Modificar Relacionamento" (Figura 5.8), que inicialmente disponibiliza a lista dos relacionamentos existentes no BD. Através dessa lista, o usuário escolher o relacionamento que vai ser modificado (rel). Escolhido o relacionamento, é exibida uma lista com os objetos da classe origem do relacionamento. Através dessa lista, o usuário escolhe um objeto origem (Objeto1), sendo exibidas, automaticamente, duas listas de objetos da classe relacionada, uma com os objetos já relacionados ao objeto origem e outra com objetos que podem ser associados ao objeto origem escolhido.

Com a lista de objetos já relacionados, o usuário escolher um objeto formando o par <objeto_origem, objeto_relacionado> que será modificado. Através da lista de objetos que podem ser relacionados, o usuário escolhe o novo objeto (Objeto3) que será relacionado ao objeto origem depois da modificação do relacionamento.

Figura 5.8 Janela para alterar um relacionamento.

Os parâmetros passados por esta operação ao "monitor" são: o nome da operação, o nome do relacionamento e os identificadores dos objetos afetados pela operação, ou seja, os objetos envolvidos no antigo e novo relacionamento.

f) GR_INSERT

Esta operação é utilizada para inserir um objeto em um grupo.

A operação recebe como parâmetros: o nome do agrupamento (classe grupo), o grupo (instância da classe grupo) e o objeto que será incluído no grupo, ou seja, **"Gr_insert(ObjetoID, grupo, classe_grupo)"**.

Figura 5.9 Janela para inserir um elemento em um grupo.

A operação é realizada através da janela "Inserir em Grupo" (Figura 5.9), que inicialmente disponibiliza a lista das classe grupos, classes que definem agrupamentos, existentes no BD. Através da lista, o usuário escolher o agrupamento onde se deseja inserir um objeto. Em seguida, duas listas são exibidas, uma com os grupos (instâncias da classe grupo) e outra com os objetos da classe elemento do agrupamento. A partir da lista de grupos e objetos, o usuário escolhe o objeto e em que grupo ele será incluído.

Os parâmetros passados ao "monitor" são: o nome da operação, o nome da classe grupo e o identificador do objeto inserido.

g) GR_DELETE

Esta operação é utilizada para excluir um objeto em um determinado grupo.

A operação recebe como parâmetros: o nome do agrupamento (classe grupo), o grupo (instância da classe grupo) e o objeto que será excluído do grupo, ou seja **"Gr_delete(ObjetoID, grupo, classe_grupo)"**.

A operação é realizada através da janela "Excluir de Grupo" (Figura 5.10), que inicialmente disponibiliza a lista das classes grupos existentes no BD. Com esta lista, o usuário escolhe o agrupamento de onde deseja excluir um objeto. Em seguida, é exibida uma lista com as instâncias da classe grupo, ou seja, os grupos que formam o agrupamento. Através da lista, o usuário escolhe o grupo que terá um componente excluído. Depois que o grupo é selecionado, uma nova lista com os objetos existentes no grupo é criada, possibilitando a escolha do objeto que será excluído do grupo.

Figura 5.10 Janela para excluir um elemento de um grupo.

Os parâmetros passados ao "monitor" são: o nome da operação, o nome da classe grupo e o identificador do objeto excluído.

5.2 - O SISTEMA GeRATOM

A partir desta seção, descrevemos o GeRATOM mostrando sua estrutura e detalhando cada um de seus componentes.

5.2.1 – ARQUITETURA DO GeRATOM

O GeRATOM basicamente é constituído pelos seguintes módulos:

- Gerenciador de Regras (GR). Utilizado para realizar operações para a manutenção da base de regras, por exemplo, incluir uma regra, excluir um evento, incluir um novo conceito;

- Monitor de Eventos. Detecta a ocorrência dos eventos e sinalizando-os ao "Processador de Regras";

- Processador de Regras. Processa as regras disparadas em decorrência do acontecimento de um evento;

- Gerenciador de Operações (GOp). Gerencia a execução das operações de modificação sobre o BD, realizando a operação ou cancelando sua execução. O GOp é o intermediário entre o MO, Processador de Regras e o BD.

Um esquema da estrutura e o fluxo de mensagens entre os módulos do GeRATOM é mostrado na figura 5.11.

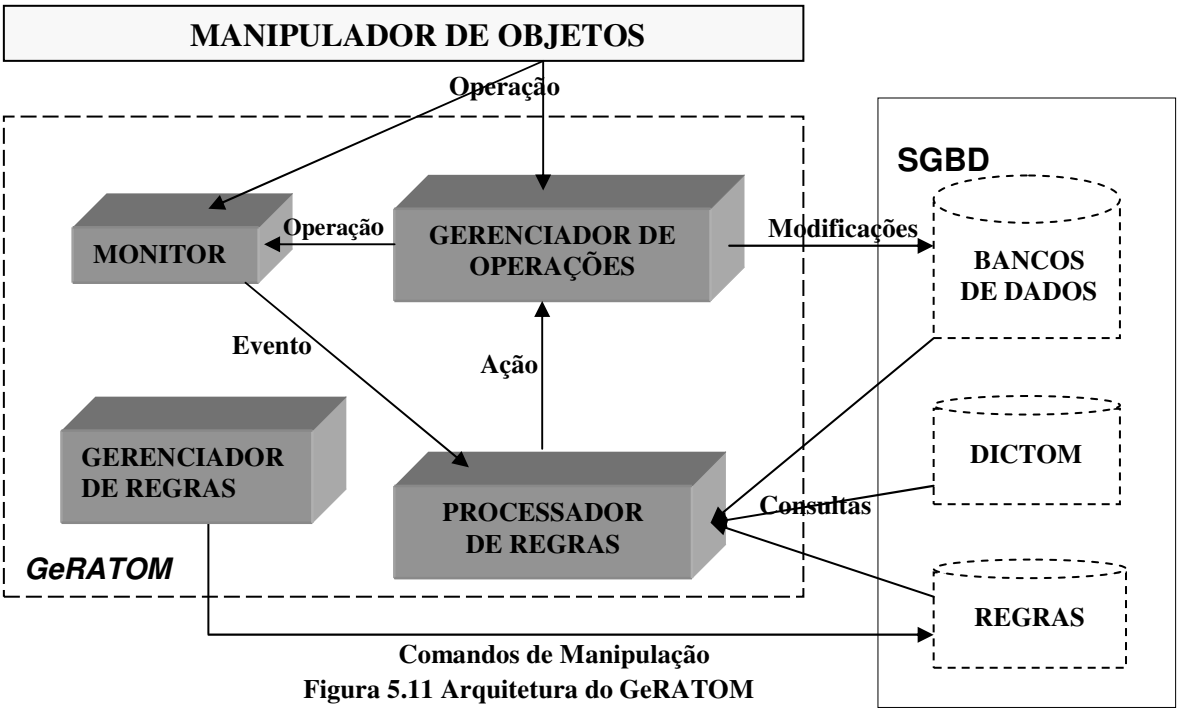

Figura 5.11 Arquitetura do GeRATOM

5.2.2 - INTERAÇÃO ENTRE OS MÓDULOS DO SISTEMA

Durante a execução de uma operação elementar, os módulos do GeRATOM e o Manipulador de Objetos interagem para evitar que a operação leve o BD a um estado inconsistente.

Quando um usuário deseja realizar uma modificação sobre o BD, ele solicita a execução de uma operação elementar através do Manipulador de Objetos (MO).

Depois que o usuário solicitar a execução de uma operação elementar, o MO passa informações sobre a operação ao Gerenciador de Operações (GOp) e ao Monitor de eventos.

O GOp armazena temporariamente a operação, até que todas as regras associadas à operação sejam processadas, enquanto que o Monitor, sinaliza ao Processador de Regras (PR) a ocorrência do evento, iniciando assim, o processamento dos ADs da operação.

O Processador de Regras recupera, da base de regras, as regras que devem ser disparadas, realizado em seguida seu processamento. Durante o processamento dos ADs e ECs serão consultadas informações do DICTOM e do próprio BD.

Se algum AD da operação for violado, o GOp recebe a sinalização para executar a operação 'Abort', como ação**,** cancelando a operação elementar.

Por outro lado, caso a operação não seja cancelada, são processados os ECs da operação. Durante o processamento dos ECs, a medida que suas condições são avaliadas como verdadeiras, as intenções de suas ações são passados ao GOp, que por sua vez, sinaliza ao Monitor as intenções das ações, iniciando o processamento de seus ADs e ECs.

Finalizado o processamento dos ECs da operação, o GOp executa a operação original e as ações dos ECs sobre o BD.

Então, podemos verificar que as operações no GeRATOM não são executadas diretamente sobre o BD, mas são agrupadas e realizas em conjunto, caso nenhuma operação do conjunto viole alguma restrição.

Vale salientar que as ações dos ECs também são operações elementares, portanto, é necessário realizar todo o processamento, descrito anteriormente, para cada ação executada.

A seguir, descrevemos em maiores detalhes os módulo do GeRATOM.

5.3 – GERENCIADOR DE REGRAS

O "Gerenciador de Regras" (Figura 5.12) permite realizar operações de manipulação dos elementos relacionados ao controle de integridade. As operações são realizadas através de um conjunto de janelas com funcionalidades bem definidas.

Figura 5.12 Janela inicial do Gerenciador de Regras

O menu principal do Gerenciador de Regras (GR) permite escolher o objeto e, em seguida, a operação de manipulação que se deseja realizar sobre o objeto escolhido. As opções do menu são as seguintes:

Regra. Permite escolher as operações de manipulação de regras;
Evento. Permite escolher as operações de manipulação de eventos;
Gatilho. Permite escolher as operações de manipulação de gatilhos;
Conceito. Permite escolher as operações de manipulação de conceitos;
AD/EC. Permite escolher as operações de manipulação de AD/EC;
Predicado. Permite escolher as operações de manipulação de predicados estruturais;
Ligação. Permite escolher as operações referentes à ligação dos AD/EC com os conceitos ou com as classes de uma aplicação;
Sair. Encerra a execução do GR.

A definição de regras, eventos, gatilhos, conceitos e outros elementos referentes ao controle de integridade, são armazenados em um conjunto tabelas relacionais, denominado Base de Regras. Portanto, todas as operações de manipulação no GR são executadas sobre a Base de Regras. A Base de Regras é composta pelas seguintes tabela:

REGRAS (Rule-Name , Priority);
EVENTOS(Event-Name, Active, Signalled, Type);
EVENTOBD (Event-Name , Method, Class);
REGRA_EVENTOS (Rule-Name, Event-Name);
GATILHOS(Trigger-Name, Action, F-Action , Condition , Alias, Priority, Status, Seq-Exec);
REGRA_GATILHOS(Rule-Name, Trigger-Name)
ELEMENTO_REGRAS (Nome-BD, Elemento, Regra)
PREDICADO(Nome-Predicado, Parâmetro, Tipo, Categoria);
CONCEITO_REGRAS(Conceito, Regra)
MODELO_CONCEITO(Conceito, Tabela-Conceito)

No apêndice B, descrevemos detalhadamente cada tabela da Base de Regras.

A seguir, descrevemos como o GR permite executar suas operações de manipulação.

5.3.1 - MANIPULAÇÃO DE REGRAS

Inserir Regra

A inclusão de uma nova regra é feita através da janela "Inserir Regra" (Figura 5.13) e consiste na criação de uma nova instância da classe REGRA.

As regras são identificadas por um nome e não é permitido mais de uma regra com o mesmo nome no sistema. Para incluir uma regra deve-se especificar seu nome, os eventos que disparam a regra e os gatilhos que serão disparados pela regra.

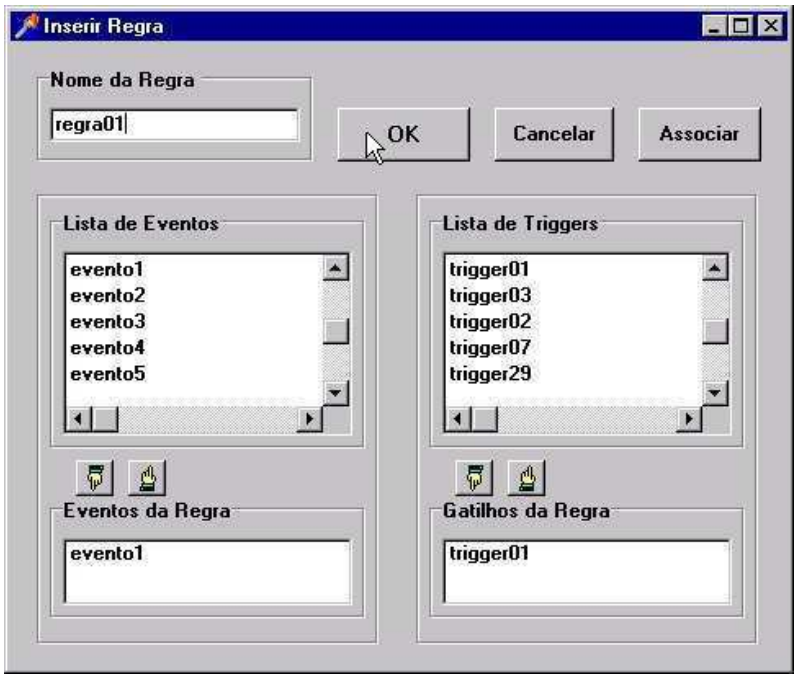

Figura 5.13 Janela para inserir uma regra

A escolha dos eventos e gatilhos são feitas através das listas denominadas 'Lista de Eventos' e 'Lista de Triggers'.

Cada instância de evento tem a propriedade Of-rules que determina quais regras são disparadas pelo evento. Então, no momento da criação de uma regra, além da sua inclusão propriamente dita, realizamos a atualização da propriedade Of-rules dos eventos da nova regra.

Se a regra que está sendo criada é uma regra de aplicação, faz-se necessário associa-la diretamente a uma classe da aplicação para permitir que a regra seja processada pelo sistema. Esta operação é realizada selecionando-se o botão "Associar" da janela "Inserir Regra". Em seguida, a janela "Associar Regra" (Figura 5.14) será exibida para permitir a escolha da classe que a nova regra será associada. Depois que a regra é criada, o relacionamento regra-classe é estabelecido.

Figura 5.14 Janela para associar regra à classe

Excluir Regra

A exclusão de uma regra consiste na eliminação de uma instância da classe REGRA. A operação é refletida na propriedade "Of-rules" dos eventos associados à regra excluída, ou seja, a referência a regra que foi excluída é eliminada das propriedades Of-rules dos eventos que disparavam a regra.

A exclusão de uma regra é feito através da janela "Excluir Regra" (Figura 5.15) e o único parâmetro necessário é o nome da regra.

Figura 5.15 Janela para excluir uma regra

Modificar Regra

Na implementação atual do GeRATOM, a modificação de uma regra é feita através da exclusão de regra que se deseja modificar e inclusão de uma nova regra com as modificações realizadas.

5.3.2 - MANIPULAÇÃO DE EVENTOS

Em relação aos eventos é possível incluir, excluir e modificar seu status (habilitar/desabilitar).

Inserir Evento

A inclusão de um evento consiste em criar uma instância da classe EVENTO e é realizada utilizando a janela "Inserir Evento" (Figura 5.17).

Para inserir um evento são necessários:

9) o nome do evento, que deve ser único no sistema;
10) o nome da operação que terá sua execução monitorada para sinalizar o ocorrência de um evento.

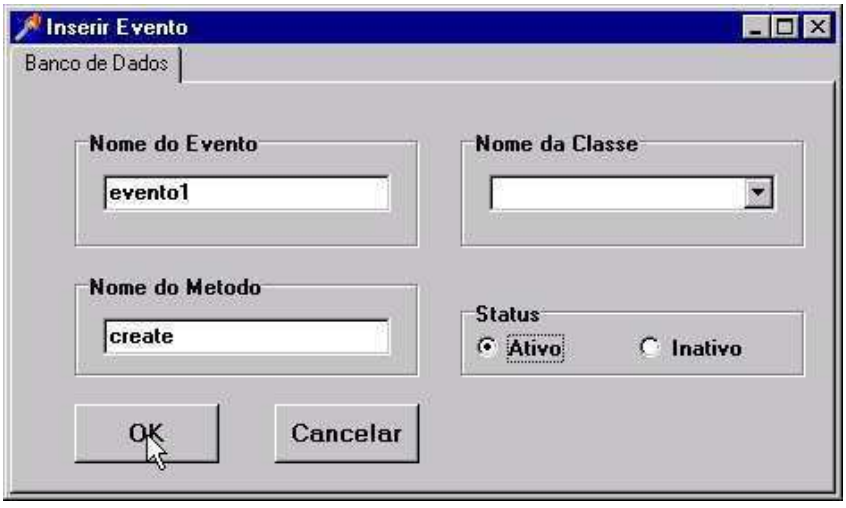

Figura 5.17 Janela para inserir um evento

Excluir Evento

A exclusão de um evento consiste na eliminação de uma instância da classe EVENTO. A exclusão de um evento é feita através de uma janela similar a janela 'Excluir Regra (Figura 5.16) e o único parâmetro necessário é o nome do evento.

Só será permitida a exclusão de um evento se nenhuma regra estiver a ele associada.

Modificar Status do Evento

A modificação do status de um evento, determina se o evento deve ou não ser monitorado pelo sistema, em outras palavras, se ele deve ou não ser passado ao Processador de Regras pelo Monitor.

Esta operação é realizada através da janela "Modifica Status" (Figura 5.18) e os parâmetros necessários são o nome do evento e seu novo status.

Figura 5.18 Janela para modificar status de eventos

5.3.3 - MANIPULAÇÃO DE GATILHOS

As operações de manipulação de gatilhos são incluir, excluir e modificar algumas de suas propriedades como: status, prioridade, condição e ação.

Inserir um gatilho

A inclusão de um novo gatilho é feita através da janela "Inserir Gatilho" (Figura 5.19) e corresponde a criação de uma instância da classe GATILHO.

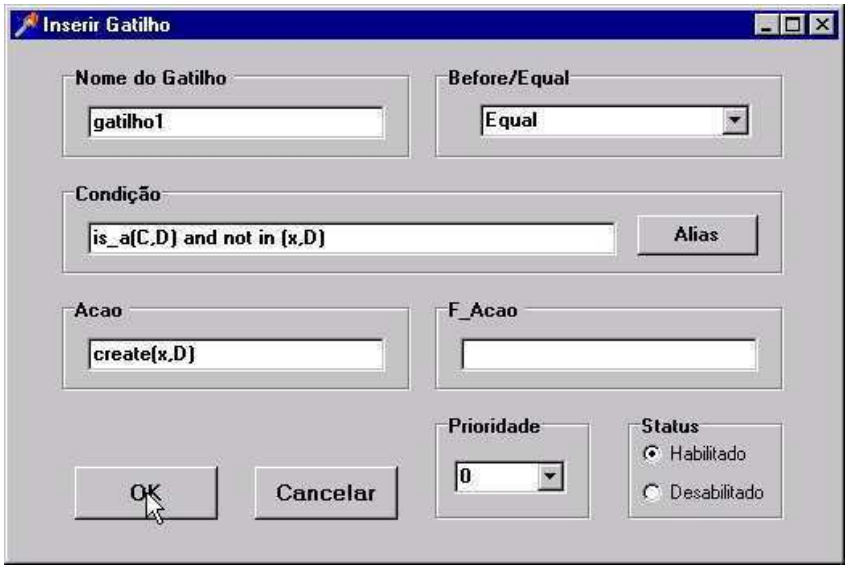

Figura 5.19 Janela para inserir um novo gatilho

Para inserir um gatilho são necessários:

- nome do gatilho, que deverá ser único entre os gatilhos do sistema;
- A condição de avaliação do gatilho. Sua especificação é opcional;
- Os *aliases* da condição para permitir a passagem dos parâmetros do evento para a condição. Os *aliases* determinam quais variáveis da condição irão receber os parâmetros passados pelo evento;
- A ação que será executada se a condição for verdadeira.
- A f-ação que será executada caso a condição seja falsa. A especificação da f-ação também é opcional;
- Determinar a prioridade de execução do gatilho;
- Determinar se o gatilho estará habilitado ou não depois da inclusão;
- Determinar se o gatilho será disparado antes ou depois da operação que disparou sua regra.

Excluir Gatilho

A exclusão de um gatilho consiste em eliminar uma instância de GATILHO. Esta operação também é realizada através de uma janela similar a janela ‘Excluir Regra” (Figura 5.16) e o único parâmetro necessário para a execução da operação é o nome do gatilho.

Só será permitido excluir um gatilho se ele não estive associada a nenhuma regra do sistema.

Modificar um gatilho;

Em um gatilho é possível modificar sua prioridade, seu status, sua condição, sua ação e a sua f-ação. Estas operações são realizadas através da janela "Modificar Gatilho" (Figura 5.20). E os parâmetros necessários são o nome do gatilho e o novo valor da propriedade que esta sendo modificada.

Figura 5.20 Janela para modificar gatilho

5.3.5 - MANIPULAÇÃO DE PREDICADOS ESTRUTURAIS

Inserir Predicado Estrutural

Um predicado estrutural é um procedimento que verifica os dados ou a definição das aplicações segundo as características estruturais de um conceito do modelo e são utilizados para especificar a condição das regras do modelo.

Para que um procedimento caracterize um predicado estrutural, é necessário que ele seja cadastrado no GeRATOM. Esta operação é realizada através da janela "Adicionar Predicado" (Figura 5.21) e consiste basicamente em incluir os dados do novo predicado no cadastro de predicados do sistema.

Para realizar a criação de um novo predicado são solicitados: o nome do predicado, ou seja, o nome do procedimento que implementa o predicado e os parâmetros do procedimento com seus respectivos tipos (string ou inteiro) e categoria (IN e OUT). Todo procedimento, para ser utilizado como um predicado, obrigatoriamente, deve retornar verdadeiro ou falso.

Figura 5.21 Janela para inserir um novo predicado

Na atual implementação do GeRATOM, a especificação de novos predicados esta limitada à criação de *stored procedures*.

Em uma *stored procedure* que implementa um predicado estrutural, o parâmetro de retorno devem assumir apenas dois valores: 1 (um) indicando que o predicado retornou verdadeiro e 0 (zero) indicando que o predicado retornou falso, ou seja, ele é do tipo "Inteiro" e da categoria "Out". Por exemplo, a declaração de uma *stored procedure* que define um predicado pode ser Is_rel(x IN STRING, y IN STRING, z OUT INTEGER), onde Is_rel é o nome do predicado e da *stored procedure*, x e y são parâmetros exclusivamente de entrada e z é o parâmetro de retorno do predicado.

O número de parâmetros de categoria IN não tem limite, enquanto que só pode existir um parâmetro da categoria OUT por predicado, utilizado pelo "Processador de Gatilhos" para verificar o resultado do processamento do predicado durante a avaliação de uma condição.

Excluir Predicado Estrutural

A exclusão de um predicado consiste em retirar do cadastro de predicados do sistema a referência ao procedimento que o implementa e, como a maioria das

operações de exclusão no GR, é realizada através de uma janela similar a janela "Excluir Regra" (Figura 5.16). O único parâmetro solicitado pela operação é o nome do predicado.

5.3.5 - MANIPULAÇÃO DE CONCEITOS

Para garantir a característica aberta do TOM é necessário que o sistema de regras reconheça os novos conceitos criados no modelo. Para isso, o GR permite incluir e excluir novos conceitos, a fim de garantir o processamento adequado de seus ADs e ECs.

Inserir um conceito

A medida que novos conceitos são criados no TOM, eles devem ser inseridos no GeRATOM para permitir o processamento das regras que modelam seus ADs e ECs. A inclusão de um novo conceito é feita através da janela "Inserir Conceito" (Figura 5.22) e consiste em cadastrá-lo no sistema. O parâmetro necessário para incluir um conceito é o seu nome.

Figura 5.22 Janela para inserir um novo conceito

A inclusão de um novo conceito é indispensável ao processamento das regras do modelo, pois elas são associadas diretamente aos conceitos e não às classes das aplicações.

Excluir Conceito

A exclusão de um conceito consiste em eliminar o conceito do cadastro de conceitos do sistema. A operação é realizada através de uma janela similar a janela

'Excluir Regra" (Figura 5.16) e o parâmetro necessário para sua execução é o nome do conceito.

Não será permitida a exclusão de um conceito se existir uma classe que o utilize em sua definição.

5.3.6 - MANIPULAÇÃO DE AXIOMAS DINAMICOS E EFEITOS COLATERAIS

O GR permite incluir e excluir um AD ou EC, associar uma regra a um conceito, caracterizando-a como uma regra do modelo, e associar as regras do modelos às classes de um BD.

Os ADs e ECs são modelados através de regras, portanto, as operações de manipulação de regras, descritas em 3.5.1, são válidas para a sua manipulação. Quanto as operações associar uma regra a um conceito e a associar regras do modelo às classes das aplicações, descrevemos a seguir.

Associar AD ou EC a um conceito

Para uma regra caracterizar um AD ou EC, ela deve ser associada a um conceito. Esta tarefa é realizada através da janela "Associar Regra/Conceito" (Figura 5.23) e consiste na inclusão de um registro com o nome da regra e o nome do conceito, na tabela "Conceito_Regras".

Figura 5.23 Janela para associar regra e conceito.

Para realizar esta tarefa são necessários: o nome da regra e o nome do conceito.

Associar ADs e ECs às classes;

Esta operação é indispensável para o processamento das regras do modelo pelo GeRATOM, pois ela cria relacionamentos entre as classes das aplicações e as regras do

modelo (AD e EC), permitindo assim, identificar quais regras devem ser processadas quando uma operação modificar uma determinada classe.

Durante a criação dos relacionamentos, são recuperadas do DICTOM os nomes de todas as classes que utilizam um determinado conceito em sua definição. Em seguida, os ADs e ECs do conceito são associados às classes recuperadas.

Esta operação é realizada através da janela "Associar AD/EC ao BD" (Figura 5.24) e o único parâmetro necessário é o nome do BD. Os relacionamentos criados pela operação são armazenados na tabela "Elemento_Regras" da base de regras.

Figura 5.24 Janela para associar AD/EC a um BD

5.4 - MONITOR DE EVENTOS

Quando uma determinada operação vai ser ou foi realizada, informações a seu respeito são enviadas ao Monitor de eventos. Baseado nas informações da operação, o Monitor verifica se a operação caracteriza um evento para o GeRATOM. Se a verificação for verdadeira, o Monitor sinaliza a ocorrência do evento ao Processador de Regras, que inicia o processamento das regras disparadas pelo evento.

5.5 - PROCESSADOR DE REGRAS

O Processador de Regras (PR) recebe do Monitor a sinalização do evento, recupera as regras associadas ao evento, realizando em seguida, o processamento de cada regra recuperada.

O PR pode ser visualizado de acordo com a seguinte estrutura:

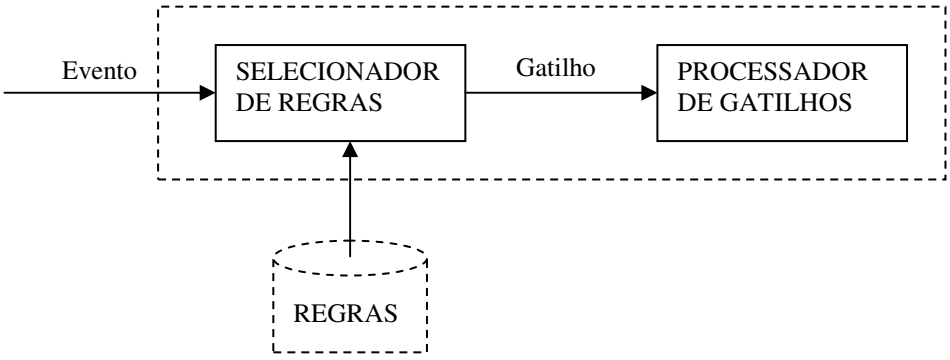

Figura 5.25 Arquitetura interna do Processador de Regras

Citamos anteriormente que no GeRATOM, as definições de regras e gatilhos são armazenadas em tabelas relacionais. Então, antes de iniciar o processamento de uma regra ou de um gatilho, sua definição é recuperada da Base de Regras e, de acordo como os dados recuperados, as classe Regra e Gatilho são instanciadas.

O PR reage a sinalização de um evento da seguinte forma:

No momento que o Selecionador de Regra (SR) recebe a sinalização de um evento, ele recupera a definição das regras disparadas pela ocorrência do evento da base de regras. Em seguida, cria as instâncias da classe Regra de acordo com as definições recuperadas.

Para cada regra, são recuperadas as definições de seus gatilhos. As instâncias dos gatilhos são criadas e associadas a regra através da propriedade 'Triggers'. A recuperação dos gatilhos é feita de acordo com a prioridade de execução dos gatilhos, ou seja, gatilhos com maior prioridade são recuperados primeiro.

Baseado na propriedade 'Triggers', os gatilhos são passados ao Processador de Gatilhos (PG) que cria instâncias das classes Condição e Ação para cada gatilho, inicia a avaliação da condição e executa a ação do gatilho se condição retornar TRUE, caso contrário, executará a f-ação.

5.6 - GERENCIADOR DE OPERAÇÕES

Toda modificação em um BD TOM é feita através da execução das operações elementares do modelo, que possuem um conjunto de regras associadas que são disparadas em decorrência de suas execuções.

Algumas regras são verificadas antes da execução da operação (ADs) verificando se a operação viola alguma restrição sobre os dados do BD. Se a violação acontecer, a operação é cancelada. Mas, se nenhum AD for violado, são processados os ECs que executam operações complementares à operação que disparou as regras para restaurar a consistência do BD.

A execução de uma operação elementar e das ações de seus ECs não podem ser dissociadas, uma vez que os ECs recompõem a base de dados a um estado consistente com a definição do BD. Portanto, se uma das operações complementares não puder ser executada, as operações complementadas devem ser desfeitas.

Outra característica dos ECs é que suas ações também são operações elementares, consequentemente, novas operações elementares serão executadas em decorrência de suas execuções, o que implicará na necessidade de realizar todo o controle de integridade sobre as novas operações realizadas.

Pelo descrito nos parágrafos anteriores, podemos considerar que a execução de uma operação elementar e das ações de seus ECs, formam um conjunto indivisível de operações que é executado como um todo ou é cancelado.

No GeRATOM temos o módulo Gerenciador de Operações (GOp) que garante o cancelamento da operação original e das ações complementares, caso uma ação complementar venha a ser cancelada em conseqüência da violação de alguma regra de integridade.

Para conseguir este processamento durante a execução dos ADs e ECs das operações elementares, todo o processamento das regras do modelo é feita em relação à intenção de execução de uma operação elementar e não em relação ao resultado de sua execução.

Com o GOp, nenhuma operação é realizada diretamente sobre o BD. Sua intenção é armazenada no Log_Operação, estrutura utilizada para armazenar as intenções de execução de uma operação e das ações das regras associados à operação. Para cada solicitação de execução de uma operação elementar, originada do Manipulador de Objetos (operação original), é criada uma estrutura Log_Operação, que armazenará as intenções da operação e das ações de todas as regras disparadas pela operação e que tiveram sua condição de execução avaliadas como verdadeira (operações complementares).

Para cada intenção de operação armazenada no Log_Operação, o GOp passa informações da operação ao Monitor, que detecta o evento e o sinaliza ao Processador de Regras, iniciando assim, o processamento da regras associadas à operação.

A medida que as ações das regras vão sendo executadas, suas intenções são armazenadas no Log_Operação e suas regras são processadas.

O disparo de regras continua até que todas as intenções tenham sido consideradas ou uma intenção viole alguma restrição. Se nenhuma operação for cancelada, o GOp utiliza o Log_Operação para executar todas as operações sobre o BD. Mas, se ocorre a violação de uma restrição todas as operações armazenadas no Log_Operação são canceladas.

Este modelo evita o uso indevido de valores já alterados no BD, mas que eventualmente serão desfeitos por uma operação 'Abort' e garante a terminação do processamento dos ADs e ECs de uma operação elementar.

5.7 - TERMINAÇÃO

Com o tratamento de intenções para as operações, descritos na seção anterior, é possível evitar a geração de eventos repetidos, evitando o disparo sucessivos de regras.

Toda vez que a condição de um EC é satisfeita, sua ação é armazenada na estrutura Log_operação e, posteriormente seu evento é gerado. Então, para evitar a geração de eventos já gerados, antes da inclusão de uma intenção de execução, é verificado se a operação já existe na estrutura. Se a verificação for verdadeira, a inclusão é bloqueada.

Um caso de geração de eventos repetidos, em decorrência da execução de uma operação elementar, é descrito a seguir.

Considere os efeitos colaterais da operação 'Delete' em uma generalização.

EC3 ⇒"*Um delete não deve violar uma generalização covering*"

ON *x delete (C)* IF *is-a (C, D)* $\wedge$ *covering (D, C_1,...,C_n)* $\wedge$ *not in_subclasse(x, C, D)* DO *x delete (D)*

EC6 ⇒ "*Ao ser eliminado um objeto de uma classe genérica, ele deve ser eliminado em todas as subclasses*"

ON *x delete (D)* IF *is-a (C, D)* $\wedge$ *in_subclasse(x, A, C)* DO *x delete (C)*

Supondo que temos a seguinte generalização *covering* (Figura 5.26),

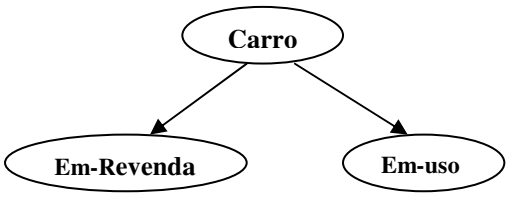

Figura 5.26 Generalização Covering

Considerando que desejamos excluir um objeto com identificação '52' da classe "Em_uso". Então a intenção da operação é inserida no Log_Operação e seus ADs e ECs são processados.

Nº Operação	**Intenção da Operação**
01	Delete (52, Em_uso)

O primeiro EC considerado é o EC3, cuja ação é excluir o objeto '52' da superclasse, caso o objeto não pertença a outra subclasse diferente da classe 'Em_uso'.

Supondo que o objeto 52 não pertence a outra subclasse, então a ação do EC3 deverá ser realizada, ou seja sua ação deve ser incluído no Log_Operação. O novo estado do Log_operação é o seguinte

Nº Operação	**Intenção da Operação**
01	Delete (52, Em_uso)
02	Delete (52,Carro)

Finalizado o processamento dos ECs da operação 01, passamos a processar as regras da operação 02. Considerando a intenção da operação e os dados do BD, que continuam com os dados de antes da execução da operação 01, a condição do EC6 será satisfeita e a operação de exclusão do objeto 52 da subclasse 'Em_uso' é incluída no Lop_operação.

Nº Operação	**Intenção da Operação**
01	**Delete (52, Em_uso)**
02	Delete (52,Carro)
03	**Delete (52, Em_uso)**

Nesse momento, temos a inclusão de uma operação já considerada, o que provocará a geração repetida de um evento, causando o processamento cíclico de um conjunto de regras.

Com a estratégia de verificar se uma intenção já foi incluída e proibir a inclusão repetida da operação, evitamos a criação de ciclo de regras dessa natureza. Para o caso descrito, ao detectar que a Op3 é igual a Op1, a inclusão de Op3 é cancelada, o que evita a geração repetida do evento associado à realização de Op3 e Op1.

Neste capítulo mostramos que combinando as funcionalidades do GeRATOM, do Manipulador de Objetos e o DICTOM criamos um ambiente capaz de processar os ADs e ECs com toda a flexibilidade deseja para um suporte adequado ao controle de integridade dos dados no TOM.

Com relação as regras de aplicação, seu processamento é similar ao das regras do modelo, com a diferença que as regras de aplicações deverão ser associadas explicitamente às classes definidas nas aplicações, diferente das regras do modelo que são associadas explicitamente aos conceitos.

Na implementação atual, o GeRATOM permite a especificação de regras em relação a um conjunto de eventos simples, mais precisamente em relação as operações elementares do TOM, o que já permite o controle da semântica dos dados considerando os conceitos do modelo.

Ainda considerando os eventos suportados pelo GeRATOM, ele permitem a especificação de regras semânticas inerentes às aplicações, mas, com uma abrangência limitada, uma vez que as regras só podem ser especificadas em relação às operações elementares.

Para um controle completo de integridade dos dados de uma aplicação é necessário que ele disponibilize meios para a especificação de regras em relação a um grupo mais abrangente de eventos, por exemplo, eventos temporais absolutos, relativos, periódicos e eventos compostos (E1 and E2).

Um outro aspecto que deve ser ressaltado sobre o GeRATOM é que o sistema foi implementado com o objetivo de permitir o processamento dos ADs e ECs dos conceitos predefinidos e de possíveis novos conceitos do TOM. Na atual implementação o sistema é capaz de processar os ADs e ECs dos conceitos predefinidos pois todas as informações para esta atividade são disponibilizadas no DICTOM e os predicados estruturais foram implementados considerando um conjunto de regras de mapeamento predefinido (seção 5.1.2).

Para permitir que o GeRATOM consiga manter a integridade dos dados segundo novos conceitos do TOM é necessário que as informações referentes a estes novos conceitos sejam disponibilizadas de forma similar às informações dos conceitos predefinidos, em outras palavras, devem ser definidas, segundo os novos conceitos, regras de mapeamento TOM-Relacional, estruturas no DICTOM para armazenar as informações de classes que utilizam os novos conceitos, implementar os predicados estruturais dos novos conceitos segundo as regras de mapeamento e as informações do DICTOM, capacitar os mapeadores a suportar as novas regras de mapeamento definidas e a gerar as informações adequadas para serem armazenadas no DICTOM.

6 – CONCLUSÕES E PROPOSTAS PARA TRABALHOS FUTUROS

6.1 – CONSIDERAÇÕES FINAIS

O GeRATOM é um sistema desenvolvido para controlar a integridade dos dados de um banco de dados descrito segundo o modelo TOM. No sistema foram inseridas todas as restrições de integridade dos conceitos predefinidos do modelo, o que já garante um mecanismo para a definição e implementação de bancos de dados orientados a objetos temporais.

Considerando a característica de modelo aberto do TOM, o sistema é capaz de reconhecer um novo conceito definido no modelo e permitir a inclusão de novas regras do modelo ao sistema, garantindo assim, não só a integridade dos dados, segundo os conceitos predefinidos, mas também, em relação aos novos conceitos do modelo.

Outra característica do sistema é que sua implementação levou em consideração o padrão SQL-92 o que permite, com algumas restrições, migrar o GeRATOM de um SGBD relacional base para outro.

As restrições do sistema referem-se aos predicados estruturais definidos pelo usuário que são implementados utilizando *stored procedures* do SGBD base. Caso o SGBD não permita chamadas externas a suas *stored procedures* ou a passagem de resultados de suas execuções a aplicações definidas fora do SGBD, o GeRATOM garantirá apenas a integridade dos dados definidos utilizando os conceitos predefinidos do modelo.

Felizmente, a maioria dos SGBDs permitem a execução de *stored procedures* através de chamadas externas, como também, a disponibilização de seus resultados para uso pelas aplicações fora do SGBD, o que garante a manutenção da integridade dos dados, considerando os novos conceitos do modelo.

Pelo discutido até o momento, podemos afirmar que o GeRATOM é capaz de manter a integridade dos dados definidos no TOM considerando os conceitos predefinidos do modelo, como também, os possíveis novos conceitos que venham a ser inseridos no modelo.

Com relação a um novo conceito é necessário criar uma tabela no DICTOM para manter as informações de definição das classes segundo este novo conceito, capacitar os mapeadores a realizar o mapeamento correto de tais classes segundo o novo conceito e gerar as informações de suas definições para serem armazenadas no DICTOM; informações estas que serão utilizadas durante o processamentos dos ADs e ECs do novo conceito.

Outra característica do GeRATOM é que agregado a um SGBD relacional simples, o sistema permite a modelagem de aplicações complexas, sem que o projetista precise utilizar artifícios para representar a semântica das aplicações no BD, estreitando dessa forma a distância entre a modelagem conceitual e o modelo interna das aplicações.

6.2 – PROPOSTAS PARA TRABALHOS FUTUROS

Para dar continuidade ao trabalho sugerimos:

- Fazer um estudo do desempenho do sistema para melhor o processamento das regras.
- Fazer um estudo a respeito da terminação das regras;
- Acrescentar aos mapeadores TOM/ODMG e ODMG/SQL a capacidade de absorver novos conceitos do modelo;
- Considerar o tratamento de versões de objetos no sistema;
- Implementar as funcionalidade da primeiro versão do TOM-Rules não implementadas no GeRATOM, por exemplo, o processamento de eventos temporais;
- Desenvolver um modelo de transações para operações conceituais da aplicação. Com transações, os ECs terão que levar em consideração, antes de sua execução, se suas ações já foram realizadas pelo próprio usuário dentro de uma transação;
- Automatizar a construção e a atualização do DICTOM para que o dicionário consiga espelhar a nova composição do modelo depois de uma modificação, ou seja, inclusão ou exclusão de um conceito.

BIBLIOGRAFIA

[1] Alhir, S.S. – "UML in a Nutshell" – O'Reilly & Associates – Sebastopol – 1998.

[2] Burns, T. – "Reference Model for DBMS Standardization" - ACM SIGMOD 15(1) - 1986 (19-58).

[3] Bauzer, C.; Pfeffer, P. – " Transformação de um Banco de Dados Orientados a Objetos em um BD Ativo" – Anais do 6º Simpósio Brasileiro de Banco de Dados – Manaus – 1991.

[4] Carvalho, A.F. – "TOM-Rules – Um Monitor de Eventos, Regras e Gatilhos em um Ambiente Orientado a Objeto" – Dissertação de Mestrado – COPIN/UFPB – Campina Grande – 1993.

[5] Ceri, S.; Widom, J. - "Deriving Production Rules for Constraints Maintainance" – Proceedings of VLDB 90 – Brisbane – Austrália – 1990 (566-577).

[6] Ceri, S.; Widom, J. - "Active Database Systems – Triggers and Rules for Advanced Database Processing" – Morgan Kaufmann, San Francisco – 1996.

[7] Chen, P.P - "The Entity-Relationship Model – Toward a Unified View of Data" – ACM TODS, Volume 1, 1976 (9-36).

[8] Cannan, S.J.; Otten, G.A.M. –" SQL – The Standard Handbook – McGraw-Hill – London – 1992.

[9] David, M. B. – "Descrição Formal da Estrutura do Modelo Orientado a Objetos Temporais - TOM", Dissertação de Mestrado – COPIN/UFPB, Campina Grande/PB – 1992.

[10] Dayal, U; Buchmann, A.P.; McCarthi, D.R. – " Rules Are Objects Too: A Knowledge Model for na Active Oriented Database System" – Proceedings of 2nd. International Workshop on OODBS – Spring-Verlag – 1988 (129-143)

[11] Diaz, O. and Paton, N. W. – " MetaClasses in Object-oriented Databases" - Object-Oriented Database – Analysis, Design & Construction (DS-4) - North-Holand – 1991 (331-347).

[12] Elmasri, R. and Navathe, S.B. – "Fundamentals of Database Systems" – Benjamin/Cummings Publishing - Redwood City - 1992.

[13] Ferreira, A. – "Sistema de Efeitos Colaterais em THM" – Dissertação de Mestrado – COPIN/UFPB – Campina Grande – 1987.

[14] Fraternali, P.; Paraboschi, S. – "A Review of Repairing Techniques for Integrity Maintenance" – Proceedings of the 1st. International Workshop on Rules in Database Systems – Edinburgh, Scotland – 1993.

[15] Furtado, M.E.S.; Schiel, U – "Uma Metodologia para Projeto de Banco de Dados Temporal Orientado a Objetos" - Anais VIII Simpósio Brasileiro de Bancos de Dados – Campina Grande – 1993 (313-327).

[16] Hull, R.; King, R. – "Semantic Database Modeling: Survey, Application and Research Issues" – ACM Computing Survey, 19-3, 1997.

[17] Hammer, M; McLeod, D. - "Database Descriptions with SDM: A Semantic Database Model" - ACM TODS - Volume 6, No. 3 – 1981.

[18] Maia, F.A.A. – " Uma Ferramenta CASE para a Metodologia FADO" – Dissertação de Mestrado – COPIN/UFPB – 1999.

[19] Microsoft Corporation, "Microsoft SQL Server 7.0 On-line Documentation", 1998.

[20] Oracle Corporation – "Oracle8 Documentation" - Redwood City, CA - 1999.

[21] Paton, N.W. – "ADAM – Object-oriented Database System Implemented in Prolog" - Proceedings of 7Th BNCOD, CUP – 1989 (147-161).

[22] Santos, D.V. - "Transformação de Esquemas de Objetos para um Gerenciador Relacional Estendido, considerando o padrão ODMG" - Dissertação de Mestrado – COPIN/UFPB – Campina Grande/PB – 1998.

[23] Schiel, U. – "An Abstract Introduction to the Temporal Hierarchic Data Model (THM)" – Proceedings of 9th International Conference on VLDB, Florença – Itália – 1983.

[24] Schiel, U. – "An Open Environment for Objects with Time and Versioning", Proceedings of East European Conference on Object Oriented Programming, Bratislava – Czecho-Slovakia – 1991 (116-125).

[25] Schiel, U.; Carvalho, A.F. – "TOM-RULES: A Uniform and Flexible Approach to Events, Constraints and Derived Information", Anais do VIII Simpósio Brasileiro de Banco de Dados – Campina Grande/PB – 1993 (372-382).

[26] Schiel, U.; Mistrik, I. – "POKER – Petri-Net Oriented Knowledge Engineering Research" – Anais X Simpósio Brasileiro de Bancos de Dados – Recife/PE - 1995

[27] Schiel, U.; Fernandes, S.L. – "O Modelo Temporal de Objetos – TOM" – Relatório Técnico - Nº DSC 002/96 - COPIN/DSC - 1996.

[28] Fernandes. S; Schiel, U. – " Um Ambiente Gráfico de Consultas a um Banco de Dados Temporal" – Anais IX Simpósio Brasileiro de Bancos de Dados – São Carlos/SP – 1994.

[29] Stonebraker, M; Hanson, E.N.; Potamianos, S. – "The Postgres Rule Management" – IEEE Software Engineering, Volume 14, Nº 7 – 1988 (897-907).

[30] Tanaka, A K. – "Bancos de Dados Ativos" – Anais do X Simpósio de Brasileiro de Banco de Dados – Recife/PE – 1995.

[31] Urban, S. D.; Karadimce, A. P.; Nannapaneni, R. B. – "The Implementation and Evaluation of Integrity Maintenance Rules in an Object-Oriented Database" – Proceedings of 8th. International Conference on Data Engineering - Phoenix, Arizona – 1992.

[32] Zaniolo, C.; Ceri, S.; Faloutsos, C.; Snograss, R.T.; Subrahmanian, V.S.; Zicari, R. – "Advanced Database Systems – Capítulo 2" – Morgan Kaufmann, San Francisco – 1997

APÊNDICE A – LINGUAGEM DE DEFINIÇÃO DE DADOS DO TOM

Este apêndice descreve, em BNF, a sintaxe completa da (meta)linguagem do modelo TOM para a definição de classes e metaclasses. A linguagem é descrita seguinte a seguinte notação:

- expressão em **negrito** ou **entre aspas** significa símbolo terminal
- **<identificador>** significa símbolo não terminal
- **[cláusula]** significa cláusula opcional
- **(expressão)*** significa que a expressão ocorre 0 ou mais vezes
- **(expressão)+** significa que a expressão ocorre 1 ou mais vezes
- | significa ou

[meta]class <nome classe>
[***class relationships***
(<nome rel> ":" <nome classe> "(" <card-min> "," <card-max> ")")*
(***pre-class*** ":" <nome classe> [***exclusive***])*
(***post-class*** ":" <nome classe> [***exclusive***])*]

Declara que <nome rel> é um relacionamento existente entre a classe em definição como um todo e uma instância da classe relacionada. <card-min> e <card-max> especificam que cada instância da primeira classe está relacionada a no mínimo <card-min> e no máximo <card-max> instâncias da segunda classe. *Pre-class* e *post-class* declaram a pré-classe e a pós-classe da classe em definição.

[***class methods*** <def. método>+]
Declara os métodos de classe associados com a classe em definição.

[***instance relationships***
(<nome rel> ":" <nome classe> "(" <card-min> "," <card-max> ")"
[***with*** <num. positivo> (***logic*** | ***transaction*** | ***bitemporal***) ***values***])+]

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

[***instance methods*** <def. método>+]
Declara os métodos de instância associados com a classe em definição.

(***keys are*** (<chaves>+ | ***inherited***) | (***type*** <tipo>)
<chaves> é composta por um ou mais relacionamentos que identificam completamente as instâncias da classe.***Type*** declara a presente classe como uma classe de domínio (primitiva).

generalization-of <lista classes> [***by*** <nome papel>] ([***explicit***] | ***by predicate*** <predicado> | ***using*** (<relacionamento> ***as index***) [***parameters***: [***disjunctive***,] [***covering***]]

Define uma hierarquia de **generalização** tendo como classe genérica a classe em definição e, como subclasses, <lista classes>.

grouping of <nome classe> (***all*** | ***explicit*** | ***by predicate*** <predicado> | ***using*** <relacionamento>) [***parameters***: [***disjunctive***,] [***covering***,] [***ordered***]]
Declara que a classe em definição é um **agrupamento** de <nome classe>.

aggregation [***redefinition***] ***of*** (<lista classes> (***all*** | ***explicit***) | <classe1, classe2> ***by*** <relacionamento>) [***exclusive***]
Declara que as instâncias da classe em definição são compostas pela **agregação** de instâncias das classes listadas em <lista classes>. O parâmetro ***redefinition*** especifica se <lista classes> é uma redefinição da estrutura de agregação herdada das classes superiores, na hierarquia.

[***parameters***: ***with time*** [***and lifetime*** <constante> : <unidade tempo>]]
O parâmetro ***with time*** relaciona cada instância da classe a uma classe *time-interval*. O intervalo começa com o tempo de inserção e termina com o tempo atual ou, para instâncias removidas, com o tempo de remoção. Se o ***lifetime*** é usado, instâncias podem pertencer à classe somente no período de tempo especificado.

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

<def. método> ⇒ <nome método> ["(" <lista parâmetros> ")"]
<lista parâmetros> é a lista de parâmetros de entrada.
[***pre-conditions*** (<predicado> [***at*** <tempo>] [***otherwise*** (***warning*** | ***cancel*** | ***error***) [<mensagem>]] | ***let*** <variavel> ***be*** <termo>)+]
<predicado> é uma conjunção de predicados primitivos que precisam ser satisfeitos a fim de que o corpo de método (declarações) possa ser executado.
Se alguma pré-condição falha, a cláusula ***otherwise*** especifica qual ação deve ser tomada. Há três opções de ação: a) ***warning*** - a falha não evita a execução do método. O sistema apenas emite uma mensagem de advertência, b) ***cancel*** - o método não é executado, c) ***error*** - toda a cadeia de mensagens que originou a operação atual é cancelada. A cláusula ***let*** atribui o valor de <termo> à <variável>.

[***body*** (<declaração> [***only if*** <predicado>])+ | ***case*** (<predicado>:<declaração>)+ |
for each <condição> ***do*** <declaração>]
[***post-conditions*** (<predicado> [at <tempo>] [***otherwise*** (***warning*** | ***cancel*** | ***error***) [<mensagem>] | ***let*** <variavel> ***be*** <termo>)+]

Eis a definição dos termos utilizados nas cláusulas anteriores:

<condição> ⇒ <var. entidade> ***in*** (<entidade grupo> | <nome classe>) | ***class*** <var. classe>) [***such that*** <predicado>]
<declaração> ⇒ <op. primitiva> | <mensagem> | ***let*** <variavel> ***be*** <termo>
<op. primitiva> ⇒ (<create> | <delete> | < establish> | <remove> | <move> | <update> | <gr-insert> | <gr-delete>) [***at*** <tempo>]

<mensagem> ⇒ <identificador> <chamada método>
<chamada método> ⇒ <nome-método> ["(" <lista parâmetros> ")"]
<create> ⇒ <nome classe> ***create*** "(" <termo> ")" [***at*** <tempo>]
<delete> ⇒ <termo> ***delete*** ["(" <nome classe> ")"]
<establish> ⇒ (<termo1> | <nome classe> | ***new***) ***establish*** "("<rel-nome> ","
<termo2> ")" [***at*** <tempo>]
<remove> ⇒ "<" (<termo1> | <nome classe>) ["," <rel-nome> "," <termo2>]
">" ***remove***
<move> ⇒ <termo> ***move*** "(" <nome classe1> "," <nome classe2> ")"
<update> ⇒ <termo> | <nome classe> ***update*** "(" <relacionamento> "="
<termo> ")"
<gr-insert> ⇒ <entidade grupo> ***gr-insert*** "(" <termo> ")"
<gr-delete> ⇒ <termo> ***gr-delete*** ["(" <entidade grupo.> ")"]
<tempo> ⇒ ***clock***. <unidade> <op. aritmético> <num. positivo> | <unidade> ":"
<valor> | <tupla tempo> | "<" <tempo> ***to*** <tempo> ">"
<unidade tempo> ⇒ "dia" | "mês" | "ano" | "hora" | "minuto" | "segundo"
<tupla tempo> ⇒ <ano><mês><dia><hora><minuto><segundo>
<predicado> ⇒ <predicado primitivo> | (<termo1> <op. relacional> <termo2>) |
not <predicado> | ***if*** <predicado1> ***then*** <predicado2> |
<predicado1> ***or*** <predicado2> | <termo1> <identificador>
<termo2> "(" <termo1> "," <termo2> ")"
<predicado primitivo> ⇒ <in> | <is-rel> | <is-a> | <is-part> | <is-elem>)
<in> ⇒ <termo> ***in*** <nome classe>
<is-rel> ⇒
<is-a> ⇒ <nome classe> ***is-a*** <nome classe>
<is-part> ⇒ <nome classe> ***is-part*** <nome classe>
<is-elem> ⇒ <nome classe> ***is-elem*** <nome classe>
<termo> ⇒ <constante>) | <variável> | <função> | <termo1> <op. aritmético>
<termo2> | "-" <termo>
<identificador> ⇒ <variável> | "(" <variável> ")" | <constante>
<variável> ⇒ <var. entidade> | <var. classe>
<constante> ⇒ <número> | <string>
<var. entidade> ⇒"x" | "y" | "z"
<var. classe> ⇒ "X" | "Y" | "Z"
<entidade grupo> ⇒ <nome> | <constante>
<função> ⇒ <nom-rel> "(" <termo> ")"
<op. aritmético> ⇒ "+" | "-" | "*" | "/"
<op. relacional> ⇒ "<" | ">" | "≤" | "≥" | "=" | "≠"
<card-min> ⇒ <num. positivo>+
<card-max> ⇒ <num. positivo>+ | "*"
<chaves> ⇒ <nome rel>*
<tipo> ⇒ "integer" | "real" | "string" | "boolean"
<lista classes> ⇒ <nome classe>+
<lista termos> ⇒ <termo>+
<lista parâmetros> ⇒ <nome > | <constante> | <lista parâmetros> "," <nome > |
<lista parâmetros> "," <nome > | <lista parâmetros> ","
<constante>

<nome metaclasse> ⇒ <nome >
<nome classe> ⇒ <nome >
<nome rel> ⇒<nome >
<nome método> ⇒ <nome >
<nome papel> ⇒ <string>
<nome operação> ⇒ <nome >
<nome parâmetro> ⇒ <nome >
<nome > ⇒ <letra> | <letra> <dígito>
<letra> ⇒ "A" | "B" | "C" ..... "Z"
<digito> ⇒"0" | "1" | "2" ..... | "9"
<número> ⇒ [<sinal>] <digito>+
<num. positivo> ⇒ <digito>+
<string>⇒ <letra>+ | <letra>+ <dígito>+

APÊNDICE B – DESCRIÇÃO DA BASE DE REGRAS

A seguir, detalhamos cada tabela da base de regras do GeRATOM mostrando sua utilidade dentro do sistema o sistema e descrevendo cada um de seus atributos.

REGRAS (Rule-Name , Priority);

Objetivo: Armazenar as definições das regras do sistema;

Atributos:

Rule-Name, nome da regra;

Priority, prioridade de execução da regra.

EVENTOS(Event-Name, Active, Signalled, Type);

Objetivo: Armazenar as definições dos eventos do sistema;

Atributos:

Event-Name, nome do evento;

Active, indica se o evento deve ou não ser monitorado pelo sistema;

Signalled, indica se o evento esta sinalizado ou não;

Type, determina o tipo do evento (BD, Temporal, etc).

EVENTOBD (Event-Name , Method, Class);

Objetivo: Armazena informações específicas sobre eventos de bancos de dados

Atributos:

Event-Name, nome do evento;

Method, método cuja execução gera o evento;

Class, classe que contém o método.

REGRA_EVENTOS (Rule-Name, Event-Name);

Objetivo: Relaciona uma regra a seus eventos, corresponde a propriedade Events da classe REGRA ou Of-Rules da classe EVENTO.

Atributos:

Rule-Name, nome da regra;

Event-Name, nome do evento.

GATILHOS(Trigger-Name, Action, F-Action , Condition , Alias, Priority, Status, Seq-Exec);

Objetivo: Armazena as definições dos gatilhos do sistema

Atributos:

Trigger-Name, nome do gatilho;

Action, ação do gatilho;

F-Action, f-ação do gatilho;

Condition, expressão da condição de execução do gatilho;

Alias, variáveis da condição que receberão os parâmetros da operação que gerou o evento;

Priority, prioridade de disparado do gatilho em relação aos outros gatilhos de sua regra;

Status, determinada se o gatilho deve ou não ser disparado (0-desabilidade/ 1-habilitado)

Seq-Exec, determina se o gatilho de se disparado antes ou depois da operação que gerou o evento que disparou sua regra (Before/Equal).

REGRA_GATILHOS(Rule-Name, Trigger-Name)

Objetivo: Determina os gatilhos que são disparados por uma regra, corresponde a propriedade Triggers da classe REGRA.

Atributos:

Rule-Name, nome da regra;

Trigger-Name, nome do gatilho;

ELEMENTO_REGRAS (Nome-BD, Elemento, Regra)

Objetivo: Determina as regras que são disparadas em decorrência da modificação de uma classe ou relacionamento em um banco de dados.

Atributos:

Nome-BD: Nome do banco de dados;

Elemento: Nome de uma classe ou relacionamento;

Regra: Nome da regra;

PREDICADO(Nome-Predicado, Parâmetro, Tipo, Categoria);

Objetivo: Armazena as definições dos predicados estruturais definidos pelo usuário, corresponde ao catálogo de predicados do sistema.

Atributos:

Nome-predicado, nome do predicado estrutural;

Parâmetro, nome do parâmetro do predicado;

Tipo, tipo de dado do parâmetro (string/integer);

Categoria, função do parâmetro antes ou depois da execução do predicado (In – parâmetro de entrada/ Out – parâmetro de retorno).

CONCEITO_REGRAS(Conceito, Regra)

Objetivo: Armazena os relacionamentos entre as regras do modelo e os conceitos

Atributos:

Conceito, nome do conceito;

Regra, nome da regra.

MODELO_CONCEITO(Conceito, Tabela-Conceito)

Objetivo: Utilizada no momento de associar as regras do modelo às classes de um BD. É utilizada para obter o nome de uma visão definida para recuperar, do DICTOM, as classes que utilizaram um determinado conceito em sua definição.

Atributos:

Conceito, nome do conceito;

Tabela-Conceito, nome da visão que recupera as classes definidas com o conceito.