Universidade Federal da Paraíba

Centro de Ciências e Tecnologia

Coordenação de Pós-Graduação em Informática

Dissertação de Mestrado

# TOM Rules

## Um Monitor de Eventos, Regras e Gatilhos em um Ambiente Orientado a Objetos

por

André Felipe de Carvalho e Silva

Campina Grande, março de 1993

Universidade Federal da Paraíba

Centro de Ciências e Tecnologia

Coordenação de Pós-Graduação em Informática

André Felipe de Carvalho e Silva

# TOM Rules - Um Monitor de Eventos, Regras e Gatilhos em um Ambiente Orientado a Objetos

Este trabalho foi apresentado à Pós-Graduação em Informática do Centro de Ciências e Tecnologia da Universidade Federal da Paraíba como requisito parcial para obtenção do grau de Mestre em Informática.

Orientador Ulrich Schiel

Campina Grande, março de 1993


S586t Silva, Andre Felipe de Carvalho e
Tom rules : um monitor de eventos, regras e gatilhos em um ambiente orientado a objetos / Andre Felipe de Carvalho e Silva. - Campina Grande, 1993.
76 f.

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

1. Ambiente Orientado a Objetos 2. Modelo de Dados 3. Dissertacao I. Schiel, Ulrich, Dr. II. Universidade Federal da Paraiba - Campina Grande (PB)

CDU 004.41(043)

TOM RULES - UM MONITOR DE EVENTOS, REGRAS E GATILHOS EM UM AMBIENTE ORIENTADO A OBJETOS

ANDRÉ FELIPE DE CARVALHO E SILVA

DISSERTAÇÃO APROVADA EM 29.03.1993

PROF. ULRICH SCHIEL, DR.
ORIENTADOR

Maria de Fatima Q. V. Turnell
PROFª MARIA DE FÁTIMA Q.V. TURNELL, PH.D
COMPONENTE DA BANCA

Marcus S. Aquino
PROF. MARCUS SALERNO DE AQUINO, M.Sc
COMPONENTE DA BANCA

PROFª SONIA SCHECHTMAN SETTE, DRA.
COMPONENTE DA BANCA

CAMPINA GRANDE, 29 DE MARÇO DE 1993

## Resumo

Este trabalho dedica-se ao estudo de um modelo de dados com características ativas que objetiva extender o modelo temporal orientado a objetos TOM, buscando uma maior flexibilidade, abrangência e simplicidadede na modelagem dos aspectos do mundo real.

O modelo TOM Rules é sugerido como uma alternativa para extender o modelo TOM garantindo-lhe características ativas. O modelo divide os aspectos dinâmicos em regras, eventos e gatilhos, tratando-os como objetos comuns. O modelo suporta modelagem de regras ativas, restrições de integridade e conhecimento derivado.

O modelo proposto é detalhado e descrito formalmente. Em seguida: 1) a implementação do subsistema TOM Rules é detalhada; e 2) a interface construída para simplificar a modelagem de aspectos dinâmicos no ambiente é apresentada.

## Abstract

This work proposes a data model with active characteristics that intend to extend the object oriented model TOM, bringing more flexibility, wideness and simplicity in modelling aspects of the real world.

The TOM Rules model is presented as an alternative to extend the TOM model, supplying it with active characteristics. The model divides the dynamics aspects into rules, events and triggers, treating them as commun objects. The model supports active rules, integrity constraints and derived knowledge.

The proposed model is detailed and formally described. Following: 1) the TOM Rules subsystem implementation is detailed; and 2) the interface built to simplify the modelling of dynamics aspects in the environment is presented.

# Agradecimentos

A realização deste trabalho contou com o apoio de algumas pessoas e instituições. Gostaria de agradecer especialmente a algumas dessas pessoas e instituições.

Norma, minha mãe, pelo exemplo de vida, carinho e apoio demostrado nas horas mais difíceis.

Paulo, meu pai, pelo apoio, estímulo e orientação nos momentos decisivos.

Paulo Henrique e Ana, meus irmãos, pelo carinho e torcida demonstrado durante o tempo em que a distância nos separou.

Adriana, pelo carinho, estímulo, companheirismo e apoio técnico demostrado em todos os momentos.

Ulrich Schiel, meu orientador, pelo apoio e paciência durante o desenvolvimento deste trabalho.

Professores e funcionários do Departamento de Sistemas e Computação da Universidade Federal da Paraíba, pelo apoio que viabilizou esta pesquisa. Gostaria de agradecer em particular a Peter, Marcus Sampaio, Ana, Lilian e Zeneide.

Sílvio Meira, coordenador do Curso de Pós-Graduação em Informática da Universidade Federal de Pernambuco, por ter permitido a utilização do laboratório desta instituição em setembro de 1992.

Aos colegas Cássio e Jorge da Universidade Federal de Pernambuco, pelo apoio técnico, fundamental no aprendizado da linguagem Sather e funcionamento do gerenciador GOD.

Aos colegas da Universidade Federal da Paraíba pela colaboração e companheirismo durante os três últimos anos. Gostaria de agradecer em particular a Rossana, Ricardo, Alfredo, Glauco, Fernando(Mandi) e Jorge pelos bons momentos juntos.

Adriana e Ruth, minhas amigas de Campina Grande, pela força e hospitalidade.

# Sumário

# Lista de Figuras

# Capítulo 1

## Introdução

*Neste capítulo é feita uma explanação a respeito dos conceitos de bancos de dados ativos e orientados a objetos. Em seguida são descritos os objetivos do trabalho.*

## 1.1 Motivação

Atualmente a maior parte dos sistemas de banco de dados não permite a modelagem dos aspectos dinâmicos da informação. O desenvolvimento de bancos de dados ativos onde é permitida a execução automática de ações pré-definidas, concentrou-se em bancos de dados relacionais e dedutivos tendo sido pequena a referência na área de orientação a objetos.

A interação entre banco de dados ativos e orientação a objetos, facilitando a construção de modelos de dados que representam o mundo real, como restrições de integridade, regras e eventos temporais, motivou-nos a abordarmos uma nova opção para provimento de capacidades ativas dentro do ambiente DYNAMO [PROT92] como componente ativo do modelo de dados TOM [SCHI83], onde são propostas novas características que visam aproximar mais o modelo estudado aos requisitos do mundo real.

## 1.2 Bancos de Dados Ativos e Orientados a Objetos

Os modelos de dados surgiram a partir da necessidade de armazenar um grande volume de informações de uma maneira que venha a simplificar a manipulação, representação e processamento

dos dados. A maioria dos sistemas de banco de dados existentes suporta os modelos clássicos relacional, hierárquico ou em rede. Porém basear-se nestes modelos para projetar um banco de dados implica em algumas dificuldades em captar a semântica do mundo real com a precisão e naturalidade almejadas.

Daí surgiu a idéia de representar no banco de dados as propriedades estruturais e comportamentais da informação. A composição do paradigma de objetos com bancos de dados mostrou-se adequado para satisfação destas necessidades, pois uma importante vantagem dos conceitos de modelos de dados que formam tais sistemas é o aumento do nível semântico que pode ser captado diretamente dentro da descrição dos dados. Esta composição fez surgir os SBDOO (Sistemas de Bancos de Dados Orientados a Objetos) que numa definição simplista seria um sistema orientado a objetos com capacidade de banco de dados. Em [KHOS89] define-sa um SBDOO como um sistema com:

- Uma linguagem de consulta com capacidade de otimização de consultas.
- Suporte a persistência e transações atômicas, controle de concorrência e recuperação.
- Suporte a armazenamento de objetos complexos, índices e métodos de acesso para recuperação rápida e eficiente.

Tradicionalmente mesmo os SDBOO caracterizam-se por serem passivos, isto é, consultas e transações são executadas apenas quando for explicita a requisição para tal execução. Porém algumas aplicações requerem uma monitoração automática de condições definidas sobre o estado do banco de dados e uma capacidade de executar ações quando o banco de dados muda, isto é, requerem a modelagem de aspectos dinâmicos nos mesmos. Chamamos estes sistemas de sistemas ativos e, quando além disso, suportam persistência, classificamo-os de Banco de Dados Ativos (BDA). Para suportar estas características tanto os SBDOO quanto os modelos tradicionais necessitam de extensões conceituais e de arquitetura.

BDA são sistemas qua reagem automaticamente a eventos gerados internamente ou externamente ao sistema em si, sem a interveção do usuário, sendo esta reação pré-definida pelo usuário sobre o estado do banco de dados [BAUZ 91]

Um banco de dados deve permitir monitoramento de restrições de integridade, testes de consistência, suportar dados derivados e acesso a restrições.

A modelagem dos aspectos dinâmicos pode ser feita através de regras de alertas [CHAN82], gatilhos [GEHA91], objetos ativos [KIM90], deamons e regras [BAUZ91, DIAZ91, DAYL88b]. Dentre as regras utilizadas para modelar aspectos dinâmicos enfocaremos as regras ECA [DAYL 88a], um poderoso formalismo para definição de aspectos dinâmicos. ECA sinifica Evento-Condição-Ação. Contitui- se de um *evento* que uma vez detectado, dispara um *gatilho* que é composto de um *condição* descrevendo uma determinada situação e uma ***ação*** a ser executada caso a *condição* seja satisfeita.

Existem várias propostas de modelos, tanto a nível conceitual quanto a nível lógico, que incluem a modelagem de aspectos dinâmicos de dados, alguns baseados na filosofia orientada a objetos, outros não. Entre os modelos temos uma extensão do O2 [BAUZ91], o Ode [GEHA91], o POSTGRES [STON86], o ADAM [DIAZ91] e o HiPAC [DAYA88b], modelos surgidos para suprir as necessidades de aplicação como CIM (Computer Integrated Manfactoring), controle de tráfego aéreo, controle de processo nuclear, etc. Estas aplicações caracterizam-se por um controle descentralizado, paralelismo, distribuição inteligente, um tempo de resposta rápido e um suporte a dados não convencionais.

## 1.3 Proposta

Este trabalho dedica-se a modelagem e implementação do TOM Rules, o subsistema de regras do Temporal Object Model TOM [SCHI91]. O TOM Rules é similar as regras ECA em robustez, mas distingue-se por:

- uma separação de eventos e gatilhos, permitindo uma maior flexibilidade e reusabilidade dos eventos existentes.
- mais completo tratamento de eventos temporais que podem ser discretos, contínuos, periódicos ou relativos. Isto permite o uso do modelo para aplicações que necessitem interagir com o "clock" do sistema.
- modelagem de dados derivados onde a parte relativa a ação é uma informação derivada.

A separação de eventos e gatilhos permite uma alocação democrática de eventos primitivos para classes, sem perder a robustez e a complexidade das regras ECA, incluindo alguns eventos combinados pelos operadores lógicos "and" e "or".

A modelagem dos aspectos dinâmicos do TOM visa dar maior abrangência ao modelo, tornando mais eficiente e completa a representação do mundo real. Neste trabalho analisaremos o poder de modelagem do modelo sendo apresentada a especificação e implementação do subsistema de regras.

## 1.4 Organização do Trabalho

O trabalho está dividido em capítulos estruturados conforme descrito abaixo:

No capítulo 2 é feita uma análise sobre alguns modelos dinâmicos.

No capítulo 3 será introduzida uma breve descrição do modelo TOM em um nível informal, objetivando esboçar as características do modelo em questão.

No capítulo 4 será apresentado o modelo TOM Rules e seu meta- esquema.

No capítulo 5 será apresentada a implementação do sistema com descrição de arquitetura, estrutura de dados e funcionalidade do mesmo.

O capítulo 6 se refere a conclusões e contribuições obtidas com este trabalho.

# Capítulo 2

## Modelos Dinâmicos

*Neste capítulo são apresentados os modelos dinâmicos HIPAC, Postgres, Ode, Adam, extensão do O2, e analizado comparativamente o modo de implementação de algumas características ativas dos mesmos.*

## 2.1 Introdução

A forma com que foram implementadas as características ativas varia de acordo com o sistema. Na próxima seção abordaremos alguns modelos existentes, considerando principalmente os aspectos dinâmicos. Os modelos abordados serão:

- O2
- HIPAC
- Postgres
- Ode
- ADAM

## 2.2 O2

O O2 [BANC89,DEUX89,LECL89] é um gerenciador de banco de dados orientado a objetos baseado nos conceitos de valores complexos (das linguagens de programação) e objetos. Os objetos são encapsulados enquanto a estrutura de valores é visível e manipulável por operadores primitivos.

No O2 a informação é organizada em forma de objetos que tem um identificador e encapsula dados e comportamento. Para cada classe é associado um tipo descrevendo a estrutura dos objetos que dela fazem parte. As classes são criadas explicitamente usando comandos e são parte da hierarquia de herança. Os tipos são construidos recursivamente usando os tipos atômicos do O2 (inteiro, ponto flutuante, double, string, boolean), classes do esquema e os construtores de lista, conjunto e tupla. No O2 um conjunto de classes pode ser estruturado em hierarquia com herança, sendo inclusive suportada herança múltipla.

A manipulação dos objetos é feita através dos métodos, que são procedimentos associados às classes que definem o comportameto dos objetos dos mesmos, podendo ser públicos ou privados. No O2 a definição de métodos é feita em dois passos:

- Declaração do método através da especificação do nome do mesmo e a classe a qual ele está associado.
- Implementação do método utilizando-se de uma das linguagens de programação do O2.

O Modelo O2 suporta um conjunto de linguagens de programação, CO2 (derivada da Linguagem C) e BASICO2, um conjunto de ferramentas de geração de interfaces do usuário (LOOKS) e um ambiente de programação (OOPE). A interface padrão do sistema O2 é o seu ambiente de programação OOPE.

No O2 o usuário deve escolher entre dois tipos de organizações: *classes* cujas instâncias são objetos que encapsulam dados e comportamento; e *tipos* cujas instâncias são valores. Valores não são encapsulados, isto é, sua estrutura é conhecida pelos usuários e são manipulados por operadores.

Para cada *classe* é associado um tipo descrevendo a estrutura destas instâncias. Classes são criadas explicitamente usando comandos e são parte da hierarquia de herança.

*Tipos* não são criados explicitamente desde que eles somente apareçam como componentes das classes e não apareçam na hierarquia de herança. *Tipos* são construídos recursivamente usando os tipos atômicos do O2 (inteiro, ponto flutuante, double, string, boolean, bits), classes do esquema e os construtores de lista, conjunto e tupla.

A extensão do modelo O2 descrita em [BAUZ91] especifica regras como propriedades do esquema do banco de dados O2, associando-as as classes, métodos, ou objetos específicos, mantendo-as independentes das aplicações. Para efeito de implementação regras foram definidas como objetos, instâncias de uma classe especial embutida no O2. Deste modo as regras podem ser herdadas de uma classe específica para uma subclasse desta classe, podendo ser modificadas, inseridas e eliminadas. Além disso, regras podem ser habilitadas e inibidas para determinadas aplicações.

O sistema de gerenciamento de regras O2 tem dois níveis: *Regra Interna* e *Regra Externa*. As *Regras Externas* são mapeadas pelo sistema em *Regras Internas* de modo a esconder do usuário detalhes de implementação.

As Regras Externas são tuplas do tipo <Nome,E,C,A,T,P,S> sendo:

**Nome** - um "string" que identifica regra externamente.

**E(vento)** - expressão que descreve um conjunto de eventos que podem ativar a regra;

**C(onsulta)** - nome da consulta O2 previamente definida. A consulta especifica a condição a ser testada para executar a ação;

**A(ção)** - nome do conjunto de operações CO2 e corresponde a ação a ser executada se a condição for satisfeita;

**T(ipo)** - indica o tipo da regra. São providos dois tipos: *relacionados a mensagens* e *relacionados a tempo;*

**P(rioridade)** - é um valor inteiro utilizado para determinar a ordem de execução das regras quando um evento ocorre;

**S(tatus)** - indica se a regra está inibida ou habilitada;

O sistema O2 fornece métodos para manipulação de regras externas que podem ser invocadas dentro de programas de aplicação ou interativamente através da interface do usuário. Os métodos oferecidos são: Add, Delete, Enable, Disable, Fire (ativação de uma regra habilitada), Connect, Disconect e Change Priority, todos traduzidos em operações correspondentes de *Regras Internas*.

As *Regras Internas* constituem-se de um conjunto de objetos invisíveis ao usuário armazenados em um único conjunto de filas para um banco de dados, agrupados segundo o evento a elas associado e ordenados parcialmente segundo o campo *Priority*, que assegura que em caso de conflito dê-se primazia a execução da regra de maior prioridade. Qualquer operação em *Regra Externa* será mapeada em *Regra Interna*.

## 2.3 HIPAC

O modelo de dados HIPAC [DAYA88a] [DAYA88b] [DAYA88c] baseia- se no formalismo de regras *evento-condição-ação* (ECA). A parte *evento* de uma regra ECA especifica operações de banco de dados, eventos temporais ou sinalizações de processos arbitrários; a parte *condição* especifica consultas ao

banco de dados; e a parte *ação* especifica um programa. Quando um *evento* ocorre (é sinalizado), a *condição* é avaliada e caso satisfeita a *ação* é executada.

As regras no HIPAC são tratadas como objetos. Existe uma classe regra e toda regra é uma instância desta classe, possuindo seus próprios atributos. A diferença entre a classe *regra* e outras classes de objetos é que o HIPAC entende a semântica da regra e invoca automaticamente uma operação particular chamada *fire* que dispara a regra.

As regras são estruturalmente definidas como:

**Identificador** - como qualquer entidade, cada regra tem um único identificador;

**Evento** - causa o disparo da regra. Argumentos formais podem ser definidos para o evento;

**Condição** - o modo de ligação entre a transação gatilhada e as condições a serem avaliadas quando a regra é disparada;

**Ação** - o modo de ligação entre a avaliação da condição e a execução da ação quando a regra é ativada e a condição satisfeita;

**R. Temporais** - prioridades, valores de funcões;

**P. Contingências** - Uma ação alternativa a ser executada caso as restrições temporais não forem satisfeitas;

**Atributos** - Propriedades adicionais das regras;

Algumas propriedades das regras tem que ser especificadas e algumas são opcionais admitindo valores padronizados.

É essencial para a regra a especificação de eventos, das condições e das ações. As outras propriedades são opcionais e em caso de não especificação serão ignoradas no momento de processamento da regra.

São permitidas as seguintes operações sobre uma regra: Create, Delete, Enable, Disable, Fire.

A função Evento da regra produz a entidade *Evento*. Um evento tem um identificador e uma lista de argumentos formais. Uma operação *signal* é definida sobre o tipo da entidade *Evento* sendo executada pelo *detector de eventos* ou explicitamente por usuários ou programas de aplicação. Os eventos englobam manipulação de operações de dados, tempo e modificações externas.

A *condição* de uma regra também é um objeto sendo sua estrutura descrita por duas funções:

- modo de ligação entre a transação gatilhada (na qual o evento qua causou o disparo da regra foi sinalizado).
- uma coleção de consultas.

A função *modo de ligação* indica quando a condição deve ser avaliada em relação ao evento que ocorreu na transação gatilhada. Existem quatro possibilidades.

- *Imediatamente* quando o evento gatilhado é sinalizado. Neste caso a transação gatilhada é suspensa até a condição (e possivelmente a ação) ser executada.
- No *modo deferido*. Neste caso a condição é avaliada numa transação separada após o término da transação gatilhada.
- *Separada mas casualmente dependente*. Neste caso a condição é avaliada numa transação separada após o término da ação gatilhada. Caso a transação gatilhada "aborte" a condição não será avaliada.
- *Separada mas casualmente independente*. Neste caso a condição é avaliada numa transação separada, estando o "scheduler" livre para escalonar esta transação independentemente da transação gatilhada.

A ação de uma regra também é um objeto e sua estrutura é definida por duas funções:

- modo de ligação entre a transação na qual a condição é avaliada e a execução da ação.
- Uma operação a ser executada na ação.

O *modo de ligação* descreve se a ação deve ser executada imediatamente após a avaliação da condição; postergada até o fim da transação na qual a condição é avaliada; *separada e casualmente dependente*, ou *separada e casualmente independente*.

A operação pode ser um programa na linguagem de manipulação de dados, o que inclui operações sobre o banco de dados (recuperações, atualizações, sinalizações de eventos abstratos e outras invocações de funções sobre objetos arbitrários) ou uma mensagem para um programa ou processo externo. A operação é definida sobre argumentos que são passados pelo disparo da entidade regra (incluindo os passados por sinalização de condição). Adicionalmente as operações podem se referir ao estado do banco de dados no tempo corrente.

## 2.4 Postgres

O sistema Postgres [STON86, STON87, STON88, STON90, STON91] é uma extensão do sistema relacional Ingres. Nele foram embutidos conceitos de orientação a objetos como herança de dados, incluindo até herança múltipla. O Postgres suporta dois mecanismos para definição de tipos abstratos de dados e a utilização de procedimentos como tipos de dados.

São providos os seguintes tipos de dados no Postgres: inteiro, ponto flutuante,strings, arrays, Postquel (sequência de comandos de manipulação de dados) e procedimentos (procedimentos escritos em uma linguagem de programação de propósito geral).

Campos do tipo Postquel e procedimento podem ser utilizados para representar.

- objetos complexos ou subobjetos compartilhados através do preenchimento de um campo Postquel, com uma sequência de comandos para representar os dados de outras relações, que virão representar funções de um objeto.
- suportar múltiplas representações de dados, através da definição de um procedimento que traduza de uma representação para outra.

O sistema Postgres possui facilidades de tipos abstratos de dados (TAD), suportando ainda dados virtuais, versões de dados e dados históricos.

No Postgres as regras são definidas num sistema de regras de propósito geral a partir da sintaxe das regras de produção da segunte forma:

ON *event*(TO) *object* WHERE

POSTQUEL_qualification

THEN DO [*instead*]

POSTQUEL_commands(s)

Onde *event* se constitui de uma recuperação, modificação, remoção ou adição sobre objetos do sistema. *Object* é nome de uma classe ou coluna de classe. POSTQUEL_qualification é uma qualificação normal sem alteração. A palavra chave opcional [*instead*] indica que a açao indicada pelo POSTQUEL_commands deve ser excutada ao invés da ação que causou a ativação da regra. Caso não exista a palavra chave *"instead"* então a ação é executada em adição ao evento do usuário. POSTQUEL_commands é um conjunto de comandos Postquel com algumas modificações.

Existem duas implementações para regras: sistema de regras a *nível de registro*, e módulo de *reescritura de consulta*. O sistema de regras a *nível de registro* é chamado quando registros individuais são acessados, removidos, inseridos ou modificados. A ligação entre a ocorrência de manipulações a registros e a ativação de regras é feita através de uma marca no registro manipulado. Esta marca contém a identificação da regra correspondente e os tipos de eventos aos quais ela é sensível. Caso o usuário manipule o atributo marcado então será chamado o sistema de regras para dar prosseguimento a operação. Ao sistema de regras é passado o registro e a nova proposta. É feita uma verificação acerca da aplicabilidade da regra e caso positivo substitui-se os valores novos e correntes na parte da ação da regra e então é executada a ação. Quando a ação termina é retornado o controle ao executor, que faz a avaliação e continua. Este sistema é particularmente eficiente se existirem um grande número de regras abrangendo um pequeno conjunto de registros, caso contrário poderá ocasionar um *"overhead"* no sistema.

A segunda implementação para regras a reescritura a *nível de consulta* é a opção nos casos em que a primeira abordagem é ineficiente. Na *reescritura a nível de consulta* é feita uma conversão da regra construida pelo usuário em uma mais otimizada. As duas implementações são complementares e o sistema fornece um *módulo para escolha de regra* que sugere a melhor implementação para determinada regra.

## 2.5 Ode

O banco de dados Ode [GEHN91] é definido, consultado e manipulado utilizando-se a linguagem de programação O$^{++}$ que é uma extensão da linguagem de programação C$^{++}$ , provendo facilidades para aplicações de banco de dados.

Os objetos em O$^{++}$ são baseados nas facilidades de objetos de C$^{++}$ e são chamados *classes*. A declaração de *classes* consiste de duas partes: A especificação (tipo) e um corpo. A especificação da *classe* pode ter um campo privado armazenando informações que podem ser usadas somente pelo implementador e uma parte pública que é a interface de classe do usuário. O corpo consiste nos corpos das funções membros (métodos) declarados na especificação da classe. O$^{++}$ suporta mecanismos de herança, incluindo herança múltipla.

O modelo utilizado pelo Ode para especificar os aspectos dinâmicos distingui-se dos outros modelos estudados pela separação dos aspectos dinâmicos em *gatilhos* e *restrições*, que diferenciam-se logicamente por:

- Restrições garantem a consistência do estado do objeto (banco de dados). Quando esta consistência não pode ser mantida então a transação é *"abortada"*. Os *gatilhos* não funcionam como verificadores de consistência de objetos. Eles são ativados sempre que uma situação específica se torna verdadeira.
- As ações associadas com violações de *restrições* são executadas como parte das transações que violam as *restrições* dos objetos. Por outro lado as ações do *gatilhos* são executadas em transações separadas.
- As restrições se aplicam a objetos do momento de sua criação até o momento em que o mesmo é destruido. *Os gatilhos* têm que ser explicitamente ativados depois que o objeto for criado.

Todos os objetos de um mesmo tipo têm as mesmas *restrições,* enquanto *gatilhos* diferentes podem ser ativados por objetos diferentes, mesmo que os objetos sejam de um mesmo tipo.

A seguir detalharemos *restrições* e *gatilhos*.

*Restrições* são utilizadas para manter a noção de consistência sobre o que é tipicamente expresso utilizando tipos do sistema. Alterações que violam restrições específicas não podem ser permitidas. *Restrições* são condições booleanas associadas a definições de classe.

As *restrições* em Ode possuem duas partes: predicado e ação. A ação é executada quando o predicado não é satisfeito, sempre na mesma transação que causou a violação da restrição.

Uma restrição é especificada como se segue:

**constraint**

*constraint1 : handler1*

*constraint2 : handler2*

. . .

. . .

*constraintn : handlern*

Onde *constrainti* é uma expressão booleana que se refere a componentes da classe especificada e *handleri* é uma declaração que é executada quando a restrição é violada.

O sistema suporta dois tipos de restrições diferenciadas sob o aspecto modo de checagem: *Imediato (forte)* e *deferido (suave)*. A escolha pelos modos é feita em tempo de definição de classe. Nas

*restrição suaves* a checagem é atrasada até um instante antes do fim da transação, enquanto nas *restrições fortes* a checagem é feita no término das chamadas das funções amigas e construtores, garantindo que os objetos em nenhum instante estarão inconsistentes. Normalmente *restrições suaves* são utilizadas quando outros objetos estão envolvidos na restrição, enquanto *restrições fortes* são utilizadas quando a condição da restrição não envolve outros objetos. A diferenciação entre restrições fortes e suaves é feita através da declaração *"soft"* no início da especificação.

Os *gatilhos* como as restrições de integridade, monitoram o banco de dados para algumas condições, exceto condições que representam violações de consistência. Assim como as *restrições*, os *gatilhos* são especificados na definição das classes e consistem de uma condição e de uma ação. Os *gatilhos* são ativados por objetos específicos, são parametrizados e podem ser ativados várias vezes com diferentes valores e parâmetros. Os *gatilhos* são definidos seguindo a estrutura abaixo:

**trigger**: [perpetual] T1 *(parameter_cicle1) : trigger_body1*

[perpetual] T2 *(parameter_cicle2) : trigger_body2*

...

...

[perpetual] Tn (*parameter_ciclen) : trigger_bodyn*

*Ti* são nomes de *gatilhos*. Os parâmetros do *gatilho* podem ser utilizado em seu corpo através de uma extensão em sua definição:

*trigger_condition = trigger_action*

Especifica-se gatilhos temporais do seguinte modo:

*within expression ? trigger_condition = trigger_action*

*[ : timeout_action]*

Uma vez ativado, um *gatilho* temporal deve ser disparado dentro de um período específico (valores de ponto flutuante especificam o tempo em segundos), de outro modo a ação *time_out* será executada.

Os *gatilhos* são associados a objetos. Eles tem que ser explicitamente ativados para cada objeto, sendo esta ativação feita após a criação do objeto ou embutido o código nos construtores, através da chameda *Object_id - Ti(argumentos). Object_id é* o identificador do objeto associado ao gatilho *Ti*.

Caso um *gatilho* esteja ativo, quando sua condição torna-se verdadeira a ação associada ao *gatilho* é executada. Ao contrário das restrições que são executadas como parte da transação que viola a restrição, a ação do *gatilho* é executada numa transação separada. Uma ação *de restrição* tem que manter a integridade do banco de dados enquanto a ação do *gatilho* não tem esta preocupação.

Numa transação é limitado o número de *gatilhos*. Um *gatilho* não pode ser disparado mais de uma vez numa transação, sendo porém ilimitado o número de ativações de um mesmo *gatilho* que podem ser disparados por uma transação simples.

## 2.6 ADAM

No sistema ADAM [DIAZ91] utiliza-se *metaclasses* para prover características de orientação a objetos como reusabilidade de código e mecanismos de subclasse. *Metaclasses* não somente permitem que as classes sejam armazenadas e acessadas utilizando as facilidades do modelo de dados, como também tornam possível o refinamento do comportamento padrão na criação de classes utilizando a especialização e herança.

No ADAM eventos e regras são objetos comuns e como tal suas definições envolvem a descrição de uma estrutura (atributos) e de um comportamento (métodos).

Os eventos são definidos na classe *event_class* e nas suas subclasses *event_db, clock_event e application_event* herdando atributos e comportamento da classe hierarquicamente superior. A classe *evento_db* possui os seguintes atributos:

*OID* - identificador do objeto;

*define_method* - nome do método;

*when* - especifica o momento em que a operação ocorre;

O atributo "when" do *evento_db* pode ter os valores "before" e "after". Caso o valor seja "before" o evento será sinalizado antes da ocorrência do método e caso contrário será executado após a ocorrência do método. Este modo de representação é perfeitamente extensível, sendo possível a criação de novas classes de eventos.

A estrutura da regra é descrita pelo evento que sinaliza a regra, a condição a ser avaliada e a ação a ser executada caso a condição seja satisfeita.

A condição é um conjunto de consultas para checar se o estado do banco de dados está apropriado para a execução das ações. A ação é um conjunto de operações que funcionam como restrições de integridade, intervenção do usuário e propagação de métodos. As definições das condições e ações podem se referir ao objeto corrente ao qual a regra é aplicada e aos argumentos correntes do método que ativou a regra. Uma regra é especificada como:

| | | |
|---|---|---|
| *OID* | - | identificador da regra; |
| *event* | - | identificador do evento que ativa a regra; |
| *active_class* | - | especifica em que classe será aplicada a regra; |
| *is_it_enabled* | - | valor booleano que especifica se a regra está habilitada ou não; |
| *disabled_for* | - | especifica um conjunto de instâncias da classe especificada em *active_class* para o qual ela está desabilitada; |
| *condição* | - | conjunto de consultas ao banco de dados; |
| *ação* | - | conjunto de operações; |

Existem três subclasses de regras no ADAM: *regras definidas pelo usuário, restrições de integridade e regras de propagação*, diferenciadas pela redefinição do método *new*.

A abordagem utilizada no ADAM para ativação de regras foi um relacionamento de duas mãos entre regras e classes. Nas regras foi criado o atributo *active_class* especificando em que classe será aplicada a regra. Nas classes especificou-se o atributo *class_rules* declarando quais as regras a serem ativadas quando uma mensagem é recebida por uma instância desta classe. Deste modo indexaram-se regras por classe movendo sua herança para a hierarquia de classes. Como as regras que afetam uma instância não são somente aquelas definidas em sua classe imediata, mas aquelas definidas em suas superclasses, e para tornar mais eficiente a pesquisa de regras, foi definido como parte da estrutura de classe, o atributo *activated_by* que especifica não só as regras ativadas pelo envio da mensagem a classe corrente, mas também, as regras ativadas por todas as superclasses da mesma, evitando assim a pesquisa de regras na hierarquia de classes.

## 2.7 Descrição e Análise dos Aspectos Dinâmicos

Nesta seção será feita uma comparação entre os modelos apresentados, considerando algumas características como a linguagem de definição de aspectos dinâmicos, caracterização do modo de ativação das regras, representação de condições, tipos e funcionalidades das ações.

## 2.7.1 Linguagem de Definição

O modelo O2 suporta a linguagem CO2 uma extensão da Linguagem C criada com o objetivo de programar aplicações de banco de dados. CO2 permite definições de classes, manipulação de objetos com passagem de parâmetros e manipulação de valores através de operadores primitivos. O sistema O2 possui também uma linguagem funcional de consulta que pode ser usada interativamente ou embutida no código CO2. As regras CO2 são expressas como objetos que são instâncias de uma hierarquia especial de classes embutida no O2, sendo as operações de gerenciamento e manipulação de regras, métodos associados a classes desta hierarquia.

O protótipo HIPAC foi implementado usando SMALLTALK80, que não se limita a ser uma linguagem de programação, mas é todo um ambiente de programação no paradigma de objetos, com sofisticada interface homem-máquina, um conjunto de ferramentas como editor, compilador de código SMALLTALK para código de máquina virtual, um *"browse"* para navegação na biblioteca de classes e uma biblioteca propriamente dita com classes. As condições e ações são expressas em blocos SMALLTALK80. A execução de transações concorrentes são expressas em processos SMALLTALK80.

Postgres suporta a linguagem de consulta orientada a conjunto POSTQUEL. É possível através da linguagem definir dados procedurais, tipos de dados extendidos, regras, versões de objetos, herança, tempo, suportando ainda "aninhamento" de consultas.

O sistema Ode suporta a linguagem $O^{++}$, uma extensão compatível de $C^{++}$ que provê facilidades de criação e organização de objetos persistentes em grupos, versões de objetos, definição e manipulação de conjuntos, e associação de restrições e gatilhos a objetos.

ADAM foi implementado em PROLOG. Todos os objetos em ADAM são considerados metaclasses, classes ou instâncias. Quando o sistema é compilado a *metaclasse* chamada META_CLASS já existe. Todas as classes subsequentes são criadas através do envio da mensagem *new* a classe da qual o objeto será uma instância.

## 2.7.2 Controle de Regras

No O2 regras são consideradas propriedades do banco de dados, sendo associadas a classes, métodos ou objetos específicos, o que os torna independente das aplicações. Assim como os demais componentes do esquema, regras estão sujeitas a herança e podem ser modificadas, inseridas e removidas. Diferenciam-se dos demais componentes por ter amarrada a determinadas aplicações a habilitação ou inibição de sua ativação.

No HIPAC a estrutura de regra encapsula eventos, condições e ações. Adicionalmente regras podem ter atributos e armazenar outras informações de contexto (modos de ligação). Regras no HIPAC são objetos com os atributos *evento, condição, ação, ligação_evento_condição, ligação_condição_ação* e com as operações comuns a qualquer objeto (*criar, modificar, remover*) acrescentadas das operações especiais *fire, ativar* e *desativar.* A operação *fire* flexibiliza a utilização de regras possibilitando a ativação manual das mesmas.

No Postgres existem duas implementações para regras: sistema de regras a *nível de registro* e *reescritura de consulta.* O sistema de regras a *nível de registro* é ativado quando registros individuais são acessados, removidos, inseridos ou modificados. Particularmente eficiente se existirem um grande número de regras abrangendo um pequeno conjunto de registros. No *módulo de reescritura de consulta* o código se coloca entre o parser e o otimizador de consultas e converte o comando do usuário em uma forma alternativa priorizando a otimização.

No Ode as regras são representadas pelas construções *gatilho* e *restrições.* Estes são associados a definições de classe implementadas como rotinas das mesmas para suportarem o mecanismo de herança. As *restrições* são divididas em "fortes" e "fracas", sendo constituídas de uma expressão booleana que se refere aos componentes de uma determinada classe e de uma declaração que é executada quando uma restrição é violada. Os gatilhos são constituídos de duas partes: ação e condição, podendo ser parametrizados.

No ADAM as regras são tratadas como objetos, logo podem ser relacionadas com outros objetos como eventos e classes do sistema. A herança de regras foi movida para a hierarquia de classes através da especificação de um relacionamento entre as regras e as classes do sistema, com o objetivo de embutir numa instância todas as regras que afetam a classe a qual ela pertence, bem como todas as regras que afetam suas superclasses. O sistema suporta três tipos de classes: *regras definidas pelo usuário, restrições de integridade* e *regras de propagação.*

## 2.7.3 Eventos

No O2 as regras são ativadas quando ocorre um evento (envio de mensagem e passagem de tempo) sendo sua ativação correspondente a verificação de uma condição seguida da execução de uma ação.

No HIPAC as regras são ativadas quando um evento é sinalizado (detectado). Os eventos são expressos em álgebra de eventos e divididos em eventos associados a operações de manipulação de banco de dados (inserção, alteração e remoção), eventos associados ao "clock" do sistema e os

chamados eventos abstratos que não são diretamente detectados pelo SGBD. Estes eventos e seus argumentos são definidos no modelo, mas são sinalizados por outros sistemas e aplicações.

Os eventos no Postgres são definidos através das regras de produção podendo ser as operações new, old, recuperar, modificar, remover, aplicadas sobre nomes de classes ou colunas de classes. No caso da implementação a *nível de registro* é feita uma marca no atributo de uma instância da classe. Esta marca contém o identificador da regra correspondente e os tipos de eventos aos quais ela é sensitiva. Caso o usuário manipule o atributo marcado, será chamado o sistema de regras antes de dar prosseguimento à execução.

No Ode um evento constitui-se de qualquer manipulação de um objeto que possua associado a sua classe uma restrição ou um gatilho. Ao acessar um objeto o sistema se encarrega de checar as restrições (expressão booleana seguida opcionalmente de uma execução de uma declaração) e executar os gatilhos (avaliação de uma condição e execução opcional de uma ação), sendo a operação atômica no primeiro caso e não atômica no segundo.

O sistema de ativação de regras do ADAM é relativamente complexo. Quando ocorre um evento (envio de mensagem) o sistema faz uma pesquisa na classe a qual pertence o objeto que recebeu a mensagem, para escolher qual dentre as regras que tem sua ativação dependente da manipulação de objetos desta classe, será efetivamente ativada. Serão escolhidas as regras que possuirem como atributo "event" o identificador do evento inicialmente detectado. A ativação corresponde a checagem da parte condicional da regra e em caso de satisfação uma posterior execução da ação.

## 2.7.4 Tipos de Condições e Ações

No O2 a condição é uma consulta que retorna um valor complexo que pode ser manipulado interativamente ou por programa. Uma consulta pode ser definida sobre estados do banco de dados, classes, objetos ou valores. A ação corresponde a qualquer sequência de operações que possa ser codificada em um método O2. Tradicionalmente ações podem ser pedidos de intervenção do usuário, execução de procedimentos de banco de dados, criação de eventos que ativem outras regras (aninhamento de regras), modificação de prioridades de execução, habilitação e desabilitação de regras.

NO HIPAC a condição constitui-se numa coleção de consultas expressas numa linguagem de manipulação de dados orientada a objetos. As consultas podem se referir a argumentos de sinalização do evento. A condição será satisfeita se todas as consultas retornarem resultados não vazios. Os resultados destas consultas são passados para ação junto com os argumentos obtidos pela sinalização

do evento. A ação constitui-se de uma sequência de operações que podem vir a ser operações sobre o banco de dados ou requisições externas de programas aplicativos.

No Postgres a ação é um conjunto de comandos POSTQUEL, constituído de uma ou mais consultas e/ou atualização (adição, exclusão e modificação), já a condição é uma consulta POSTQUEL.

No Ode a condição constitui-se de uma expressão booleana e a ação é a execução de uma declaração $O^{++}$ . O que diferencia as ações nos gatilhos e nas restrições é que nos primeiros a ação é executada em uma transação separada, enquanto no segundo a execução é feita na mesma transação em que a violação de restrição é executada.

No ADAM a condição constitui-se de um conjunto de consultas que checam se o estado do banco de dados está apropriado para a execução da ação. A ação é um conjunto de operações que podem ter diferentes intuitos, como garantir restrições de integridade, intervenção do usuário, propagação de métodos, etc. As definições de condição e ação podem se referir ao objeto corrente ao qual a regra é aplicada e aos argumentos correntes do método que ativou a regra.

# Capítulo 3

## Modelo de Dados Orientado a Objetos - TOM

*No presente capítulo é discutido suscintamente o modelo TOM, e é feito um esboço do mesmo como um sistema de metaclasses, sendo feita uma breve explanação sobre a metalinguagem do modelo.*

## 3.1 Conceitos

A estrutura do modelo de dados temporal orientado a objetos TOM (Temporal Object Model) herda muitas características do modelo semântico THM (Temporal-Hierachic Data Model) [SCHI83]. Será introduzida a nova abordagem orientada a objetos nas seções a seguir.

TOM é um modelo orientado a objetos baseado nos conceitos de classe, relacionamento e método. Toda e qualquer entidade do mundo real é representada por um objeto instância de uma classe. Cada classe de objetos apresenta uma estrutura de relacionamentos e métodos que são herdados por todos os objetos daquela classe.

Toda classe tem nome e uma descrição de suas instâncias. Esta descrição é fornecida pela declaração de um conjunto de *relacionamentos-instância* (relacionamento que varia de instância para instância) e um conjunto de *métodos-instância* (utilizado numa mensagem enviada a uma instância do objeto). Os *relacionamentos-classe* associam valores à classe como um todo e servem a duas finalidades:

- considerar a classe como um objeto com seus próprios relacionamentos;
- fatorar valores comuns a todas as instâncias

Mensagens enviadas a uma classe, ao invés de a um objeto, são implementadas por *métodos-classe*. *Os métodos-classe* são usados para manipular *relacionamentos-classe*, e criar novas instâncias, ou selecionar instâncias de uma classe.

São considerados quatro tipos de classes de objetos:

- classes com objetos identificados por um ou mais relacionamentos chave;
- classes permitindo vários objetos idênticos;
- classes de instâncias absolutamente iguais (não possuem relacionamento instância).
- classes cujas instâncias são identificadas por domínios, tais como *Integer, [1..100] of Integer, Nome*.

A descrição de objetos como instâncias de classe é chamada abstração de **classificação** descrita acima, na qual os objetos são abstraídos de seus detalhes individuais e descritos por suas propriedades comuns, o modelo oferece mais três abstrações.

**generalização** - uma nova classe descreve propriedades comuns de várias classes mais específicas. O inverso é chamado especialização.

**agregação** - um objeto composto é derivado da junção de várias abstrações, sendo cada uma delas parte do objeto composto.

**agrupamento** - abstração que possibilita a definição de objetos de uma classe como conjuntos de instâncias de outra classe.

As três abstrações acima, bem como um mecanismo de herança estrutural para a hierarquia de generalização estão incorporados em muitos modelos de dados semânticos. Os modelos orientados a objetos estendem a herança estrutural da generalização para uma herança de métodos. No TOM herança é considerada também para agregação e agrupamento. Na generalização a herança é automática pois ocorre uma reconsideração do mesmo objeto do mundo real em um outro nível. Nas outras duas abstrações, novos objetos compostos são introduzidos. Devido a essa interrelação estreita e física entre o novo objeto e seus componentes, aplica-se uma herança seletiva. Identificamos três casos:

**Herança Direta** - relacionamentos são herdados diretamente pelo objeto de nível inferior sem modificações.

**Herança Computada** - os valores de algumas propriedades não podem ser herdadas diretamente. Uma computação deve ser executada.

**Nenhuma Herança** - existem relacionamentos e métodos que só se aplicam a um nível de abstração.

## 3.2 Meta-Esquema do Modelo TOM

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

TOM é um modelo de dados aberto orientado a objetos. Ele inclui um sistema de metaclasses que possibilita a descrição dos conceitos do modelo de dados, independente da aplicação. Existe uma metaclasse pré-definida denominada TOM_Class a partir da qual podem ser criadas, como instâncias, classes e metaclasses. Neste raciocínio há dois níveis de descrição: o metanível e o nível de aplicação. No metanível a metaclasse TOM_Class já está pré- definida. O projetista do modelo pode então modelar um modelo específico para uma ou várias aplicaçôes, criando novas metaclasses que representem os conceitos e abstrações do modelo. Quando o projeto de modelo de dados termina, o projetista da aplicação entra em ação criando as classes de aplicação como especialização de metaclasses apropriadas e/ou como instâncias de metaclasses definidas pelo usuário. O projetista da aplicação usa as metaclasses como conceitos do modelo necessários a um problema específico. A figura 3.1 ilustra os níveis de abstração do modelo TOM.

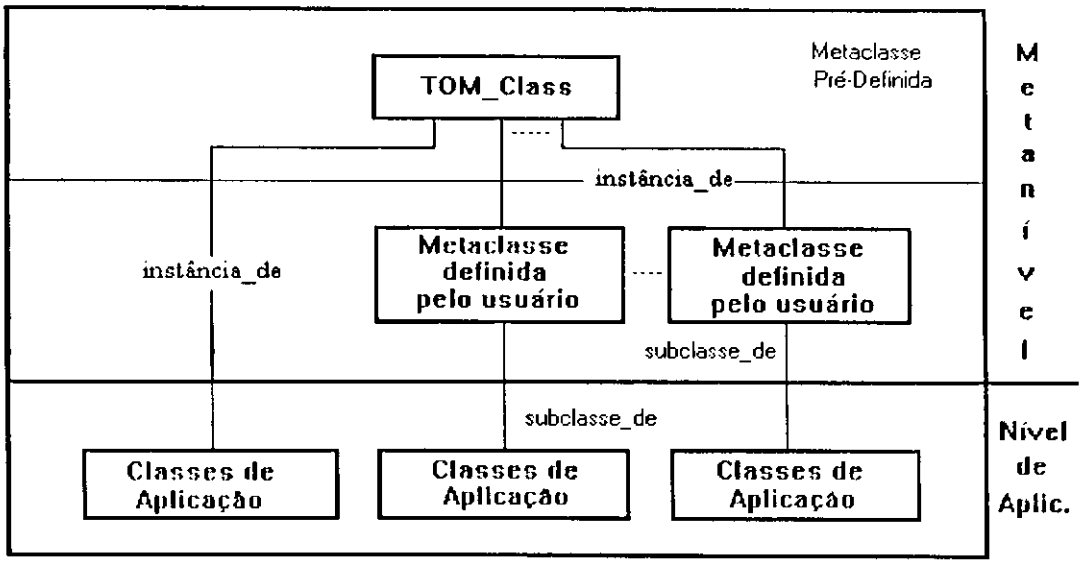


**Figura 3.1 Niveis de abstração do modelo TOM**

A linguagem utilizada para definir o esquema de modelo de dados é similar àquela usada na definição do esquema de aplicação. Usamos uma modificação da linguagem de esquema conceitual do THM [SCHI83]. A seguir será especificada e explicada a semântica da metalinguagem do modelo TOM.

## Semântica

**class** <nome classe> [**specialization** of <nome metaclasse>]
[**instance of** <nome metaclasse>]
[**class relationships:** (< nome rel> : < classe relacionada>
"(" <card_min>,<card_max >")")*
(**pre_class** ":" <classe relacionada> [exclusive])*
(**post_class** ":" <classe relacionada >[exclusive])* ]
[**class methods** <def.método>+ ]
[**instancerelationships**
(<nome rel> ":" <classe relacionada>
"(" <card_min > "," <card_max> ")"
**with** <n> (**history**| **rollback**| **temporal**) **values**] )+ ]
[**instance methods**
<def.método>+ ]
( **keys are** [<lista de chaves>+ | **inherited**)
**type** <tipo> [ format <especificação> ]
<def.agrupamento> ::=
**grouping of** <nome classe>
<def.agregação> ::=
**aggregation** [**redefinition**] **of** < lista de classes>

<def.generalização> ::=
**generalization of** <lista de classes> [explicit]
**by role** <nome papel>
{parameters: with time}
{and lifetime <constante> : <unidade tempo>}

**specialization of** <nome metaclasse>

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

**instance of** <nome metaclasse>

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

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

Declara que <nome rel> é um relacionamento existente entre a classe em definição como um todo, e uma instância da <classe relacionada>. <card min> e <card max> especificam que cada instância da primeira classe está relacionada a no mínimo <card min> e no máximo <card max> instâncias da segunda classe. Pre-class e pos-class declaram uma pré-classs e uma pós-classe da classe em definição.

**class methods** <def.método>+

Declara os métodos da classe associados com a classe em definição.

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

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

**instance methods**
<def.método>+

Declara os métodos de instância associados com a classe em definição.

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

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

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

Declara a presente classe como classe de domínio.

<def.generalização> ::=
**generalization of** < lista de classes>
**[explicit] by role** <nome papel>

Define uma hierarquia de generalização tendo como classe genérica a classe em definição e, como subclasses, <lista de classes>.

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

Declara que a classe em definição é um agrupamento de instâncias de <nome classe>.

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

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

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

O parâmetro **with time** determina que cada instância está relacionada com a classe TIME_INTERVAL classe definida no metanível do TOM. O intervalo começa com o tempo de inserção e termina com o tempo atual ou, para instâncias removidas, com o tempo de remoção. Se **lifetime** é usado, instâncias pertencem à classe somente para o período de tempo especificado.

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

<nome_método> ["(" <lista de parâmetros> ")"]
[**pre-conditions**
[<predicado>
[**otherwise**(**warning** | **cancel** | **error**)]]* ]
[ body
<declarações> [
[ **post-conditions**
[<predicado>
[**otherwise**(**warning** | **cancel** | **error**)]]* ]

onde:

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

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

<declarações> é uma sequência de operações primitivas chamadas de procedures e estruturas de controle do TOM.

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

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

"cancel" - o método não é executado.

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

A seguir será introduzida um esboço do modelo orientado a objetos TOM. No metanível são descritos os conceitos de classe, relacionamento-classe, relacionamentos-instância, métodos-classe, métodos-instância, etc. Métodos são especificados por pré- e pós- condições. Para maiores informações sobre o modelo TOM, sugere-se uma consulta a [DAVI91].

```
metaclass TOM_Class
class  relationships
       class_methods
              create(className)
                     pre-conditions not (className) in self
                     post-conditions(className) in self

instance_relationships
       class_name:CLASS_NAME(1,1)
       class_relationships:CLASS_RELATIONSHIPS(0,*)
       class_methods:CLASS_METHODS(0,*)
       instance_relationships:INSTANCE_RELATIONSHIPS(1,*)
       instance_methods:INSTANCE_METHODS(0,*)
       identification:KEY_OR_TYPE(1,1)
       generalization : ROLE_DEF (0,*)
       grouping : GROUPING_DEF (0,1)
       aggregation:AGGREGATION_OF(0,1)
       parameters : TIME (0,1)

instance_methods
       update(nome_relacionamento:string,novo_valor:DOM)
       delete()
              pre-conditions has_objects(self) = {}
              post-conditions not self in TOM_Class

identification keys are class_name

parameters with time


Metaclass CLASS_NAME
       type string

Metaclass RELATIONSHIP
       generalization of with role
              CLASS_RELATIONSHIPS,   INSTANCE_RELATIONSHIPS
```

```
        by role tipo_relação parameters covering, disjunctive
        aggregation of NOME_REL,":",CLASS_NAME,
            (CARDINALIDADE | "exclusive" | {})

Metaclass METHOD
        generalization of INSTANCE_METHODS, CLASS_METHODS
            by role tipo_uso parameters covering, disjunctive
        aggregation of
            METHOD_SIGNATURE, METHOD_IMPLEMENTATION

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


Metaclass METHOD_NAME
        type string

Metaclass METHOD_IMPLEMENTATION
        aggregation of "operation",
            METHOD_NAME,"(",PARAMETERS,")"
            METHOD_BODY

Metaclass METHOD_BODY
        aggregation of PRE_CONDITION, BODY, POS_CONDITIONS

Metaclass KEY_OR_TYPE
        generalization of
            KEY_DEF,TYPE_DEFINSTANCE_METHODS
            by role chave_ou_tipo

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

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

Metaclass AGREGGATION_DEF
        aggregation of "aggregation", ("redefinition,"|{}), "of" ("inherited"|{}),
            DEF_CLASSES, ("exclusive"|{})

Metaclass DEF_CLASSES
generalization of AGREGACOES_BINARIAS, AGREGACAO_MULTIPLA
        by role tipo_agregação

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

Metaclass LIFETIME
        aggregation of "and lifetime", CONSTANTE, ":"
            SPECIFICACAO_UNIDADE
```

# Capítulo 4

## TOM Rules

*Neste capítulo é apresentado o TOM Rules, as regras ECA do modelo orientado a objetos TOM, que descrevem regras ativas, restrições de integridade e conhecimento derivado. Em seguida é feita a especificação formal do modelo.*

## 4.1 Introdução

O modelo dinâmico TOM Rules, subsistema de regras do Temporal Object Model (TOM) [SCHI91] utilizou como base formal as regras ECA [DAYA88b,DAYA88b]. As regras ECA são uma tupla *Evento-Codição-Ação*, onde *evento* especifica operações sobre o banco de dados, eventos temporais ou sinalizações de processos arbitrários, *condição* especifica consultas sobre banco de dados e *ação* especifica um programa. Quando um *evento* ocorre (é sinalizado) a *condição* é avaliada e se satisfeita, a *ação* é executada.

O modelo TOM Rules manteve a robustez das regras ECA mas distingui-se por:

- uma separação de eventos, gatilhos e regras, através da utilização de classes distintas para representá-los. A utilização da facilidade de especialização inerente a orientação a objetos, simplifica a construção de eventos, gatilhos e regras mais específicos, que viriam a herdar características das classes originais. Esta forma de representação traz uma maior extensibilidade, flexibilidade e reusabilidade dos aspectos dinâmicos.
- mais completo tratamento de eventos temporais. Eventos podem ser discretos, contínuos, periódicos ou relativos. Isto permite o uso do modelo para aplicações que necessitem de interações com o "clock" do sistema.

A separação de eventos e gatilhos permite uma alocação democrática de eventos primitivos a classes, sem perder o poder de complexidade das regras ECA, incluindo alguns eventos combinados pelos operadores lógicos *"and"* e *"or"*.

TOM Rules são também tuplas (E, C, A) que são utilizadas para preencher três tarefas distintas em um banco de dados.

- *RegrasAtivas* - especificam ações a serem executadas sempre que certos eventos ocorrem e uma condição é satisfeita. Composto por uma parte evento/condição e uma consequente ação.
- *Retrições* - funcionam como verificadores de consistência, permitindo ou não a finalização de uma manipulação de banco de dados sempre que este evento de manipulação ocorre e uma condição é satisfeita. A parte relativa a ação tem também um componente de falha chamado *F_Ação*. Para restrições a *Ação* está vazia e a *F_Ação* pode ser uma operação específica *abort*.
- *Conhecimento Derivado* - funcionam como provedores de valores para atributos derivados. Atributos derivados são atributos de objetos que não possuem um valor imediato, necessitando que seu valor seja computado. Esta computação seria feita através de uma regra TOM Rules, composta por um evento, que seria uma manipulação do atributo derivado e um gatilho cuja ação correspondente computaria o valor do atributo em questão.

Na próxima seção introduziremos os conceitos de regras, eventos e gatilhos e ilustraremos seu uso através de exemplos.

## 4.2 Modelo de Dados

Como utilizamos um modelo orientado a objetos, as regras são especificadas como instâncias de uma classe especial do banco de dados, definidas e tratadas da mesma maneira que qualquer outro objeto, sem que seja necessário definir algum mecanismo adicional ou estrutura auxiliar. Regras são especificadas como objetos, com atributos e métodos. Portanto as características dinâmicas que as regras trouxeram foram todas implementadas através de métodos, trazendo todas as vantagens do paradigma de orientação a objetos ao gerenciamento de regras, como encapsulamento, modularidade, reusabilidade, e identificador único.

Numa abordagem uniforme o sistema não distingüe objetos comuns de regras. Como resultado as mesmas podem ser arranjadas em hierarquia e qualquer vantagem introduzida para os objetos serão

automaticamente aplicáveis às regras (mecanismos de transação, mecanismos de bloqueio, facilidades de "display").

A avaliação das regras constitui-se numa parte crítica dos sistemas dinâmicos, principalmente quando além de gerenciar atualizações no banco de dados, é feito um gerenciamento de eventos temporais. Porém a prioridade do modelo TOM Rules é maximizar a flexibilidade na modelagem aspectos dinâmicos.

As regras estão sujeitas ao controle das transações, isto é, para tratar uma regra, as transações tem que bloqueá-la para leitura, e para modificá-la ou removê-la, tem que bloqueá-la para gravação.

O subsistema de regras é baseado nas classes REGRA, EVENTO, GATILHO e MENSAGEM_EXTERNA.

## 4.2.1 Regras

A classe REGRA tem como atributos uma combinação de eventos e gatilhos. Como regras são objetos do banco de dados elas estão sujeitas a operações de banco de dados como inserir, modificar e remover regras. Além disso esta classe possui métodos específicos como ***signalRule*** que testa as pré-condições de ativação dos gatilhos e caso satisfeitas, envia uma mensagem para o objeto gatilho específico.

Esta combinação de eventos e gatilhos numa regra, especifica que um gatilho será ativado através de uma mensagem ***signalTrigger*** quando todos os eventos correspondentes tiverem sido sinalizados. Os eventos podem ser classificados em eventos de banco de dados e eventos temporais.

Eventos de banco de dados ocorrem quando uma manipulação ao banco de dados é feita. Caso seja manipulado um objeto, sendo esta manipulação previamente definida como um evento, uma mensagem será enviada à classe evento. Ela testa se a manipulação sinalizou um evento ativo e informa REGRA. Na parte relativa ao evento de uma regra, é feita uma verificação da necessidade de esperar por outros eventos. Por exemplo: Digamos que uma regra tenha sido definida como uma combinação dos eventos E1 e E2 e do gatilho T1. Se o evento E1 for sinalizado, só será disparado o gatilho se o evento E2 também for sinalizado. Uma vez sinalizados todos os eventos de uma regra, um ou mais gatilhos correspondentes serão disparados. A seguir a condição do gatilho é avaliada. Em caso de satisfação, uma ação correspondente será executada, do contrário uma ação de falha é executada.

## 4.2.2 Evento

Eventos são indicadores que sinalizam que uma situação específica foi atingida e que uma reação se faz necessária [DITT88]. No TOM Rules eventos foram definidos como objetos comuns pertencentes a classe EVENTO e às suas subclasses. Sendo assim a classe EVENTO comporta características estruturais e comportamentais comuns a todos os tipos de eventos. Objetos da classe EVENTO possuem nome, algumas regras associadas e um atributo de "status" que determina se um evento está ativo ou não. Quanto a seu comportamento, eventos possuem métodos para inserir, alterar e remover e são armazenados no banco de dados como qualquar objeto. Adicionalmente os objetos da classe EVENTO possuem o método *signal*, utilizado para sinalizar a ocorrência do mesmo, sendo executado pelo gerenciador de eventos do sistema. Informalmente definiríamos um evento como a seguir:

```
classe EVENTO
   atributos
      of_rule : set of REGRA
      nome : STRING
      ativo : BOOLEAN
   métodos
      criar -> body_criar
      alterar - >body_alterar
      remover -> body_remover
      signal -> body_signal
end
```

Eventos podem ser definidos como operações sobre o banco de dados, alteracões no "clock" do sistema e notificações externas.

Eventos definidos como manipulações sobre o banco de dados são especificados na classe EVENTO_BD, subclasse de EVENTO. Todos os objetos desta classe são sinalizados quando um método é aplicado a uma classe ou objeto ou quando uma mensagem externa chega ao sistema. Tratamos modificações no banco de dados e notificações externas da mesma maneira porque o sistema só reconhecerá sinalizações deste tipo quando receber mensagens de uma classe especial chamada MENSAGEM, definida no metanível do TOM. Para cada tipo de mensagem externa, deverá ser definida uma subclasse de MENSAGEM. Logo, qualquer evento externo será tratado como uma alteração da classe MENSAGEM e de suas subclasses, o que pode ser encarado como uma manipulação de banco de dados, tornando equivalentes o modo de especificação destes eventos e por conseguinte, similar a forma como serão tratados. Um objeto da classe EVENTO_BD além de herdar todas as características da class EVENTO, caracterizam-se por possuir dois outros atributos: *class_bd* e *method_bd*. Juntos, estes atributos especificam que aplicação de um certo método a uma certa classe, ou a um objeto de

uma certa classe configura um evento. Isto é, se definirmos um evento pertencente a class EVENTO_BD tal que:

```
classe EVENTO_BD
   atributos
      nome : "promote_employee"
      of_rule : {}
      ativo : true
      method_bd : "promote"
      class_bd : "EMPLOYEE"
end
```

Estaremos definindo que a aplicação do método "promote" a qualquer instância da classe "EMPLOYEE" constitui-se de um evento de banco de dados.

Para *eventos externos* o tratamento seria semelhante. Por exemplo: em um sistema que interage com um sensor de temperatura, onde a cada mudança de temperatura o sensor emita um sinal, poderia ser modelado através do evento abaixo.

```
classe EVENTO_BD
   atributos
      nome : "evento_temperatura"
      of_rule : {}
      ativo : true
      method_bd : "send"
       class_bd : "MENSAGEM_TEMPERATURA"
end
```

Sendo a classe MENSAGEM_TEMPERATURA uma subclasse de MENSAGEM.

Para contruirmos regras que funcionem com conhecimento derivado é necessário que os eventos que venham a compô-la pertençam a classe EVENTO_CD, subclasse de EVENTO_BD. Estes eventos, chamados de *eventos de conhecimento derivado*, são sinalizados quando é feita uma manipulação a um *atributo derivado*, que venha a ser parte de um objeto de banco de dados. *Atributos derivados* são atributos que não possuem um valor imediato, necessitando que uma computação seja feita para que seu valor seja determinado. Um objeto da classe EVENTO_CD além de herdar todas as características de EVENTO_BD, possui um atributo adicional, *atributte_bd*, que tem como domínio a classe DERIVED (que define atributos derivados), definida no metanivel do TOM. Juntos os atributos *atributte_bd, class_bd* e *method_bd* especificam que a manipulação de um atributo de objeto (especificado por *attribute_bd*) pertencente a uma certa classe (especificado por *class_bd*) por um certo método (especificado por *method_bd*) configura um evento. Caso venhamos a definir um evento da classe EVENTO_CD tal que:

**classe** EVENTO_CD
**atributos**
nome : "evento_display"
of_rule : {}
ativo : *true*
method_bd : "display"
class_bd : "STUDENT"
attribute_bd: "level"

**end**

Estaremos definindo que a manipulação do atributo *"level"* de um objeto da classe STUDENT pelo método *"display"* contitui-se de um evento de *conhecimento derivado*. O atributo *"level"* tem como domínio a classe DERIVED.

Outra subclasse de EVENTO, utilizada para modelar eventos temporais, denomina-se EVENTO_TEMPORAL. Esta classe além das características herdadas de EVENTO, possui os atributos *when* e *delay* e *tipo* utilizados para modelar intervalos absolutos no tempo. O atributo *when* é uma tupla <AA, MM, DD, hh, mm, ss> que especifica um ponto absoluto no tempo que funciona como o ponto inicial do intervalo, sendo:

- AA - dois inteiros determinantes do ano.
- MM - dois inteiros determinantes do mês .
- DD - dois inteiros determinantes do dia.
- hh - dois inteiros determinantes da hora.
- mm - dois inteiros determinantes dos minutos.
- ss - dois inteiros determinantes dos segundos .

O atributo *delay* é uma tupla da mesma forma que especifica o ponto final do intervalo de tempo. O evento ocorreria neste intervalo. Digamos que se deseje modelar um evento temporal que ocorresse em 14 de novembro de 1995 as 9:00 num intervalo de tempo de 5 segundos. Definiríamos como a seguir:

**classe** EVENTO_TEMPORAL
**atributos**
nome : "aniversário"
of_rule : {}
ativo : *true*
when : <95,11,14,09,00,00>
delay : <00,00,00,00,00,05>
tipo : *discreto*
**end**

Caso desejássemos que o evento fosse sinalizado exatamente no ponto especificado por *when* bastaria que atribuíssemos ao atributo *delay* o valor nulo. Na especificação acima atibuiu-se o valor "discreto" para o atributo *tipo*. Este atributo suporta dois valores: "discreto" e "contínuo". Quando especificado com o valor "discreto" o evento só ocorrerá uma única vez dentro do intervalo de tempo especificado em *when-delay*. Caso seja especificado como "contínuo" o evento ocorrerá durante todo o intervalo de tempo. Caso queiramos modelar o evento "em Setembro de 1993... ", definiríamos como a seguir:

**classe** EVENTO_TEMPORAL
  **atributos**
    nome : "setembro"
    of_rule : {}
    ativo : *true*
    when : <93,09,01,00,00,00>
    delay : <00,01,00,00,00,00>
    tipo : *contínuo*
**end**

Foi especificado como valor de when a tupla <93,09,01,00,00,00> (a partir de primeiro de setembro de 1993 às 00:00:00) durante um mês (delay = <00,01,00,00,00,00>) o evento "setembro" ocorrerá continuamente.

Para prover uma maior abrangência na modelagem de eventos o sistema possui duas subclasses de EVENTO_TEMPORAL, denominadas EVENTO_TEMPORAL_PERIODICO e EVENTO_TEMPORAL_RELATIVO. A primeira possibilita a modelagem de eventos periódicos, que se repetem dentro de um período específico. Além dos atributos herdados de EVENTO_TEMPORAL, em EVENTO_TEMPORAL_PERIODICO definimos o atributo *período*, uma tupla da forma < AA, MM, DD, hh, mm, ss> que especifica o período da próxima ocorrência do evento. Modelemos um evento temporal que especifique que de 1993 em diante, mensalmente, o evento venha a ocorrer.

**classe** EVENTO_TEMPORAL_PERIODICO
  **atributos**
    nome : "mensal"
    of_rule : {}
    ativo : *true*
    when : <93,01,00,00,00,00>
    delay : <00,00,00,00,00,00>
    período : <00,01,00,00,00,00>
    tipo : *discreto*
**end**

A partir de janeiro de 1993, mensalmente, o evento "mensal" ocorrerá.

A segunda subclasse de EVENTO_TEMPORAL é a classe EVENTO_TEMPORAL_RELATIVO. Esta classe permite modelar eventos que tenham seu tempo de ocorrência definidos relativamente a ocorrência de um outro evento. Modelemos um evento que ocorrerá no intervalo de um a três dias após o evento "promote_employee" modelado anteriormente:

**classe** EVENTO_TEMPORAL_RELATIVO
  **atributos**
    nome : "evento_rel"
    of_rule : {}
    ativo : *true*
    when : <00,00,01,00,00,00>
    delay : <00,00,02,00,00,00>
    evento_relativo : "promote_employee"
    tipo : *discreto*
**end**

Modelamos acima que a partir de um dia até três dias após a ocorrência do evento "promote_employee", que constitui-se de um evento de banco de dados (aplicação do método *promote* a class EMPLOYEE), o evento "evento_rel" ocorrerá. Para este tipo de evento o atributo *when* armazenará um tempo relativo, apesar de estruturalmente o atributo ter as mesmas características.

## 4.2.3 Gatilho

Um gatilho é ativado pela mensagem *fire* recebida de uma regra. É basicamente formado por uma condição e uma ação. A *condição* é um predicado em forma normal conjuntiva. Os parâmetros utilizados pelos predicados e pelas *ações* seguintes podem ser supridos por uma coleção de consultas sobre o banco de dados. A parte relativa a *ação* é dividida em duas partes: uma é executada caso a *condição* seja satisfeita e a outra caso contrário. Uma *ação* é um programa que é executado quando um *gatilho* é disparado. O corpo de uma *ação* não se limita a operações sobre o banco de dados. As *ações* são objetos pertencentes a classe ACAO (definida no metanível do TOM), possuindo métodos de inserção, modificação e execução. As *ações* são sempre armazenadas em formato executável. Através das *ações* as *regras* permitem:

- execução automática de operações de banco de dados
- verificação de restricões de integridade
- suporte para conhecimento derivado.

Sendo as ações programas, não existem dificuldades em projetá-las para manipular o banco de dados. Quanto a verificação de restrições de integridade, as regras além da capacidade de sinalizar tentativas de violação, podem reagir a tais violações através de uma ação que impede a execução da operação original, pelo envio da mensagem *"abort"* para o evento que iniciou todo o processo de disparo do gatilho.

Conhecimento derivado é implementado através de regras que compostas por eventos de conhecimento derivado. Neste caso o gatilho funcionaria como provedor de valor para o atributo derivado manipulado, que foi especificado pelo evento de conhecimento derivado. Caso o atributo venha a ser manipulado, o evento é detectado, a regra é sinalizada e a *condição* do gatilho correspondente é avaliada, sendo retornando pela *ação* do gatilho os predicados falso, verdadeiro ou um valor computado.

Como os gatilhos são tratados como objetos pertencentes a classe GATILHO, eles possuem seus próprios atributos, facilitando a criação de informações de contexto adicionais. Os gatilhos possuem métodos para criação, modificação e remoção. Possuem também os métodos *able* e *disable* de modo a tornar possível ao usuário habilitar e desabilitar manualmente o gatilho. Possui ainda o método *fire* ativado por uma mensagem recebida de um objeto REGRA correspondente, ou de um atributo de classe implícito. Os objetos pertencentes a classe GATILHO possuem os seguintes atributos:

*status* - atributo booleano que especifica se o gatilho está habilitado ou não. É um atributo privado, o que significa que seu valor só pode ser mudado pelos métodos *able e disable* que são específicos da classe GATILHO.

*prioridade* - um atributo inteiro variando de 0 a 10 que especifica a prioridade de execução do gatilho. É utilizado quando um conjunto de gatilhos são especificados como componentes de uma regra. A ordem de disparo dos mesmos obedecerá a ordenação feita sobre o atributo *prioridade* de cada gatilho. Quanto maior a prioridade mais favorecido será o gatilho durante a ordenação.

*desabilitados* - atributo que consiste numa coleção de identificadores dos objetos do banco de dados para os quais o disparo do gatilho está desabilitado. Funciona como uma espécie de tratamento de excessão.

*sequência_de_execução* - possui dois parâmetros possíveis, *before* e *equal*. Através deles é possível determinar se a ação do gatilho será executada antes ou concomitantemente com a finalização do evento. Ao especificar o parâmetro *before* como valor do atributo *sequência_de_execução*, o usuário terá definido que o gatilho funcionalmente se comportará como um verificador de consistência, sendo

bloqueado o método que sinalizou o evento. É vedada a construção de regras compostas por eventos temporais e gatilhos que funcionam como verificadores de restrição de integridade, visto ser sem sentido funcional esta combinação, pois é impossível bloquear sinalizadores de eventos temporais.

*condição* - objeto da classe CONDICAO definida no metanível do TOM, que consiste de uma coleção de predicados comparativos entre valores de banco de dados e constantes, e outros valores de banco de dados.

*ação* - conjunto de operações a serem processadas caso a condição seja satisfeita.

*f_ação* - conjunto de operações a serem processadas caso a condição não seja satisfeita. Quando o gatilho funciona como verificador de restrição de integridade pode reduzir-se ao método *"abort"*.

A seguir, objetivando tornar mais claros os conceitos apresentados, construiremos uma regra com a finalidade de modelar que quando forem feitas transferências de empregados de empresas filiais para a matriz de uma *holding,* e estas transferências coincidirem com o início do mês, sejam executados os seguintes procedimentos: requisição automática integral de *ticketes_refeição,* quando este empregado for um carregador ou possuir um salário inferior a 2000, e parcial caso contrário.

```
classe EVENTO_BD
   atributos
      nome : "trasfere_p_matriz"
      of_rule : {}
      ativo : true
      method_bd : "transfer_matriz"
      class_bd : "EMPLOYEE"
end


classe  EVENTO_TEMPORAL_PERIODICO
   atributos
      nome : "início_mes"
      of_rule : {}
      ativo : true
      when  : <93,01,01,00,00,00>
      delay  : <00,00,09,00,00,00>
      período  : <00,01,00,00,00,00>
      tipo : contínuo
end
```

```
classe GATILHO
   atributos
      nome : "Ticket"
      status : habilitado
      prioridade : 2
      desabilitado_para : {}
      seq_exec : equal
      condição : self.salary < 2000 or self.cargo = carregador
      ação : "Ticket_extra_integral"
      f_ação : "Ticket_extra_parcial"

end

classe REGRA
   atributos
      nome : "Regra_Ticket"
      eventos : {"início_mês", "transfere_p_matriz"}
      gatilhos : "Ticket"
end
```

Vale ressaltar que o atributo *of_rule* dos objetos pertencentes a classe EVENTO e subclasses inicialmente serão sempre vazios. Seu preenchimento será transparente para o projetista e ocorrerá quando for definido um objeto da classe REGRA, que especifique como valor de seu atributo *eventos* o identificador do evento em questão. No exemplo acima após a criação do objeto "Regra_Ticket" pertencente a classe REGRA, seriam atualizados os eventos componentes da regra em questão, tal que os campos *of_rule* dos eventos "início_mês" e "transfere_p_matriz" seriam preenchidos com o identificador da *"REGRA_TICKET"*. Este tratamento visa melhorar o desempenho durante a pesquisa de regras, característica que será abordada no capítulo 5.

## 4.3 Especificação Formal

Nesta seção serão especificadas as classes que modelam os aspectos dinâmicos no TOM_Rules, REGRA, EVENTO, EVENTO_BD, EVENTO_CD, EVENTO_TEMPORAL, EVENTO_TEMPORAL_PERIODICO e EVENTO_TEMPORAL_RELATIVO e GATILHO.

```
class REGRA
    class_methods
        createRule (n, t*, e*)
            pre-conditions
                not n in REGRA_NAME
                    for e in e* and for t in t*
                        e inEVENTO
                        t in GATILHO
                        if e in EVENTO_TEMPORAL then
                            t.exec_seq < "before"
            post-conditions
                new in REGRA
                new.has_name = n
                new.has_event = e*
                new.has_trigger = t*
                if e' in new.has_events then
                    e'.of_rules = new
    instance_relationship
        has_name:STRING(1,1)
        has_events : EVENTO (1,*)
        has_trigger : GATILHO (1,*)
    identification has_name
    instance_methods
        remove
            post-conditions
                not self in REGRA
        signalRule(e,x)
            pre-conditions
                ![e signal(e,o)] or
                ![e signal_ri(e,p)] or
                ![e signal(e,{})]) or
                ![e signal_cd(e,{})
            post-conditions
                for all e' in self.eventos and e <> e'
                if e' in first
                    for t' in self.trigger
                        e ![t' fire(o)] or
                        e ![t' fire_ri(p)] or
                        e ![t' fire_t]
                        e ![t' fire_cd]


class EVENTO
    class_methods
        createEvent (n, a)
            pre-conditions
                not n in EVENTO_NAME
            post-conditions
                new in EVENTO
                new.has_name = n
                new.ativo = a
    instance_relationship
        has_name:STRING(1,1)
```

```
        active : {Y,N} (1,1)
        of_rule : REGRA (0,*)
    identification has_name
    instance_methods
        remove
            post-conditions
                not self in EVENTO

class EVENTO_BD subclass of EVENTO
    class_methods
        createEvent_db (n, a, m, c)
            pre-conditions
                not n in EVENTO_NAME
                c in TOM_CLASS
                c.m in TOM_CLASS
            post-conditions
                new in EVENTO_DB
                new.has_name = n
                new.ativo = a
                new.has_method = m
                new.has_class = c
    instance_relationship
        has_method : METHOD (1,1)
        has_class : TOM_CLASS (1,1)
    instance_methods
        signal(e,o)
            pre-conditions
                [self self.has_method]
            post-conditions
                ![self.of_rules signalRule(e,o)]
        signal_ri(e,p)
            pre_conditions
                [self self.has_method]
            post-conditions
                ![self.of_rules signalRule(e,p)]

class EVENTO_CD subclass of EVENTO_BD
    class_methods
        createEvent_db (n, a, m, c,at)
            pre-conditions
                not n in EVENTO_NAME
                c in TOM_CLASS
                c.m in TOM_CLASS
                c.at in DERIVED
            post-conditions
                new in EVENTO_DB
                new.has_name = n
                new.ativo = a
                new.has_method = m
                new.has_class = c
                new.has_atribute = at
    instance_relationship
        has_method : METHOD (1,1)
        has_class : TOM_CLASS (1,1)
        has_attribute : DERIVED (1,!)
    instance_methods
        signal_cd(e,{})
```

```
                pre-conditions
                    [self self.has_method]
                    [self.has_attribute request (self.has_method)]
                post-conditions
                    ![self.of_rules signalRule(e,o)]

class EVENTO_TEMPORAL subclass of EVENTO
    class_methods
        createTemporalEvent (n, a, w, d, t)
            pre-conditions
                not n in EVENTO_NAME
            post-conditions
                new in EVENTO_TEMPORAL
                new.has_name = n
                new.when = w
                new.delay = d
                new.tipo = t
    instance_relationship
        when : TEMPO (1,1)
        delay : TIMESPAN (0,1)
        tipo : {discreto,contínuo}
    instance_methods
        signal
            pre-conditions
                now >= first.when or (now <= first.delay and first.type = "contínuo")
            post-conditions
                if first.ativo = "Y" then
                    ![first.of_rule signalRule(first,{})]
    parameters ordered by when

class EVENTO_TEMPORAL_PERIODICO subclass of EVENTO_TEMPORAL
    class_methods
        createTemporalPeriodicEvent (n, a, w, d, p, t)
            pre-conditions
                not n in EVENTO_NAME
            post-conditions
                new in EVENTO_TEMPORAL
                new.nome = n
                new.when = w
                new.delay = d
                new.periodo = p
                new.tipo = t
    instance_relationships
        when : TEMPO (1,1)
        delay : TIMESPAN (0,1)
        every : TEMPO (1,1)
        tipo : {discreto,contínuo}
    instance_methods
        signal
            pre-conditions
                now >= first.when or (now <= first.delay and first.type = "contínuo")
            post-conditions
                if first.ativo = "Y" then
                    ![first.of_rule signalRule(first,{})]
                    ![first Periodical]
        Periodical
            pre-conditions
```

```
                    let t be self.every  t < 0
                post-conditions
                    self.when = -self.when + t

class EVENTO_TEMPORAL_RELATIVO subclass of EVENTO_TEMPORAL
    class_methods
        createTemporalRelativeEvent (n, a, w, d, r, t)
        pre-conditions
            not  n in EVENTO_NAME
            r in EVENT
        post-conditions
            new  in EVENTO_TEMPORAL
            new.has_name = n
            new.when = w
            new.delay = d
            new.evento = r
            new.tipo = t
    instance_relationship
        when : TEMPO (1,1)
        delay : TIMESPAN (0,1)
        evento : EVENTO (1,1)
        tipo : {discreto,contínuo}
    instance_methods
        signal
            pre-conditions
                now >= first.when or
                (now  <= first.delay and  first.type = "contínuo" )
            post-conditions
                if first.ativo = "Y" then
                    ![first.of_rule signalRule(first,{})]
                    ![first Periodical]
        Ordenate
            pre-conditions
                ![self.eventosignalRule(self.evento)]
            post-conditions
                self.when = now + -self.when
                self.delay = now + -self.delay


class GATILHO
    class_methods
        createTrigger (n, c, a, f, s, p, d, e)
            pre-conditions
                not (n) in GATILHO_NAME
                (a)  in ACAO_NOME
                (f)  in ACAO_NOME
                (c) in CONDICAO
                (d) in TOM_OBJECT_NAME
            post-conditions
                new in GATILHO
                new.nome = n
                new.ação = a
                new.f_ação = f
                new.prioridade = p
                new.status = s
                new.condição = c
```

```
                    new.disabilitado_para = t
                    new.seq_exec = e
instance_relationship
    nome : STRING (1,1)
    ação : ACTION_NAME (1,1)
    f_ação : ACTION_NAME (0,1)
    condição : CONDICAO (0,1)
    prioridade : {0..10}
    status : {habilitado, desabilitado}
    seq_exec :{before, equal}
    disabilitado-para : TOM_OBJECT_NAME (0,*)
identification has_name
instance_methods
    fire_ri(p)
        pre-conditions
            self.status = "habilitado"
            p in STR_OB
        post-conditions
            if (self.condição = NULL) or
                (self.condição = true) then
                return(self.ação run(p))
            else return(self.f_ação run(p))
    fire_(o)
        pre-conditions
            o in TOM_OBJECT self.status = "habilitado"
        post-conditions
            if (self.condição = NULL) or (self.condição = true) then
                self.action run(o)
            else self.f_action run(o) fire_t
    fire_t
        pre-conditions
            self.status = "habilitado"
        post-conditions
            if (self.condição = NULL) or
                (self.condição = true) then
                self.action run
            else self.f_action run
    fire_cd
        pre-conditions
            self_status = "habilitado"
        post-conditions
            if (self.condição = NULL) or
                (self.condição = true) then
                return(self.action run)
            else return(self.f_action run)
    able
        pre-conditions
            self.status = "desabilitado"
        post-conditions
             self.status = "habilitado"
    disable
        pre-conditions
            self.status = "habilitado"
        post-conditions
            self.status = "desabilitado"
```

# Capítulo 5

## Implementação

*Neste capítulo é apresentada a arquitetura do sistema, com explicações a respeito de seu funcionamento e é feita uma descrição da interface com o usuário através da apresentação de todas as janelas do sistema. Em seguida é descrita a modelagem de alguns aspectos dinâmicos para demonstrar o modo de interação com a interface do sistema.*

## 5.1 Arquitetura do Sistema

O sistema TOM Rules (Figura 5.1.1) foi desenvolvido utilizando-se a linguagem orientada a objetos Sather (9500 linhas de código) e a Linguagem C (8000 linhas de código), e teve como suporte o gerenciador de objetos GOD[MEIR92] (desenvolvido na Universidade Federal de Pernambuco), e a ferramenta de construção de interface GUIDE. A maioria das estruturas foi modelada em Sather e utilizou-se C para programar a comunicação entre processos e a interface.

A utilização do gerenciador GOD deveu-se a contingência de não termos à disposição um gerenciador de banco de dados. O GOD é um gerenciador orientado a objetos que armazena informações complexas, possuindo mecanismos de recuperação de falhas, mas não é um gerenciador completo, visto que seu desenvolvimento teve o propósito de implementar apenas um subconjunto das funções de um gerenciador de banco de dados. A maior dificuldade encontrada durante a fase de implementação foi garantir a consistência entre os objetos armazenados no GOD e a especificação das classes a que pertencem estes objetos, pois o GOD funciona basicamente como um depósito de dados cujas estruturas foram definidas por usuários, e não como um servidor de estruturas e dados. Para nós esta abordagem não era a mais interessante, visto que uma vez definida uma classe, é essencial que ela esteja à disposição de todos os usuários, bem como os objetos a ela pertencentes (respeitando-se as permissões).

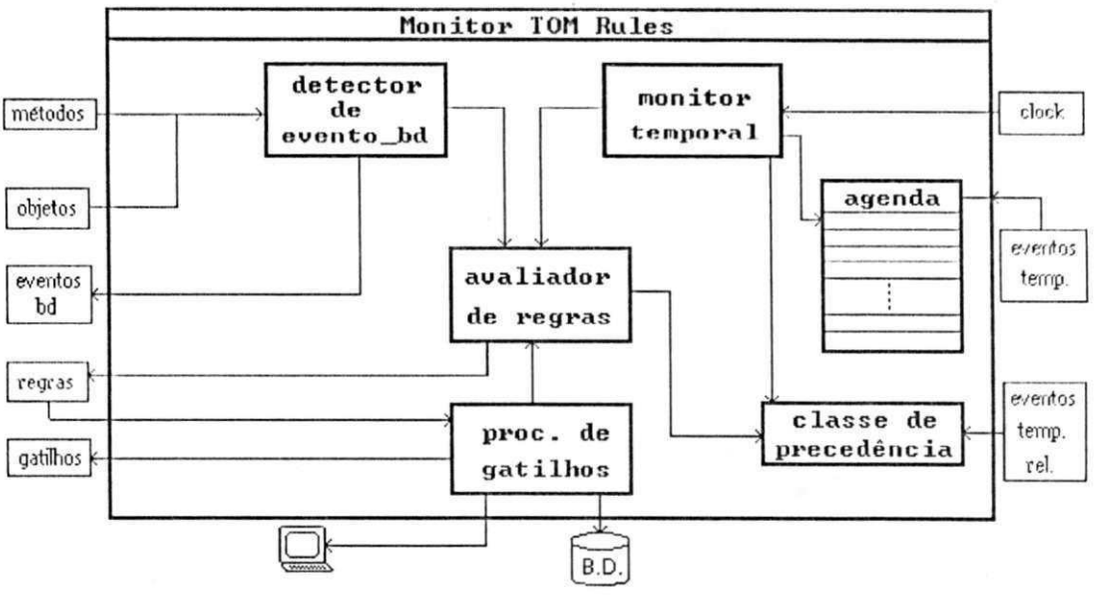


**Figura 5.1.1 Arquitetura do Sistema**

Para alcançármos estes objetivos tivemos que utilizar alguns artifícios. Limitamos a interface a ser o único meio de inserções de novas classes a serem monitoradas pelo sistema , classes estas que teriam suas estruturas armazenadas fisicamente em uma mesma localização e teriam identificadores de classe únicos. A filosofia do GOD caracteriza-se por uma não unicidade quanto aos identificadores de classe, de modo que um usuário A pode definir a classe EMPLOYEE enquanto um usuário B pode definir uma classe completamente diferente e chamá-la de EMPLOYEE. O GOD tratará ambas como versões de uma mesma classe. Em síntese, necessitamos ter o controle não somente da informação armazenada, mas também da estrutura utilizada para construí-la. Para isto, quando for instalado, o sistema criará um diretório onde serão armazenados todos os objetos monitorados, e outro onde serão armazenadas todas as especificações de classe utilizadas pelos usuários. A gravação de informações neste diretório só será feita através da interface do TOM Rules, sendo impedida através de controle de acesso qualquer outra tentativa de criação de classes.

O TOM Rules possui três módulos: inicialização, monitor e interface que serão explicados a seguir.

## 5.2 Inicialização

Para inicializar o monitor basta executar o comando "$maquina \$TOM_RULES\TOM_STARTUP inicializar < CR >", onde "$maquina" seria a máquina, "$TOM_RULES" é o diretório onde foi instalado o monitor, "TOM_STARTUP" é a chamada ao programa executável do monitor, "inicializar" o parâmetro a ser passado ao mesmo, e < CR > a indicação do acionamento da tecla "ENTER". Serão inicializadas todas as classes do sistema e criado quatro subdiretórios: GOD, CLASSES, ACOES, ACOES_FONTES. No diretório GOD serão armazenados todos os objetos monitorados pelo sistema e o arquivo executável "god_startup" (programa que ativa o GOD). No diretório CLASSES serão armazenadas todas as classes do sistema. No diretório ACOES serão armazenadas todas as ações (programas executáveis) do sistema e no diretório ACOES_FONTES os arquivos fontes destas ações. Nestes três últimos diretórios o acesso só será possível através da interface do TOM Rules, sendo esta estruturação transparente para o usuário.

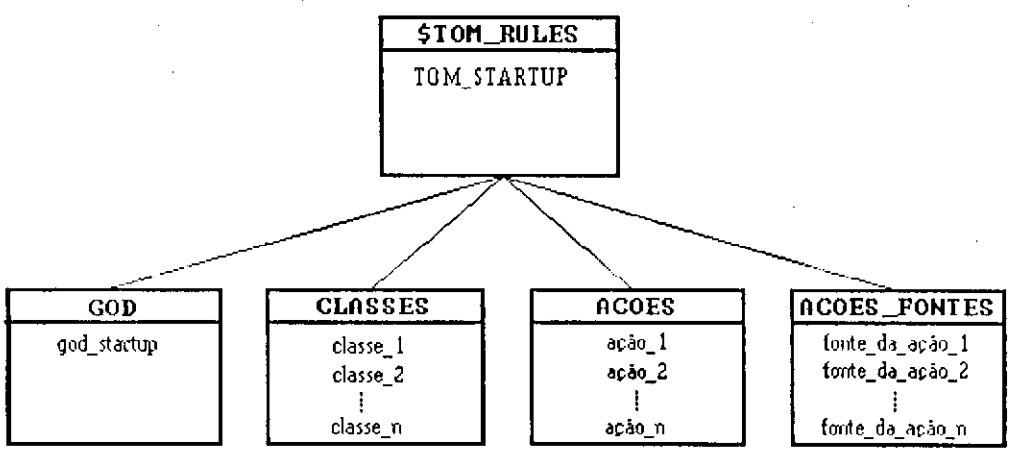


**Figura 5.2.1 Hierarquia de Diretórios**

## 5.3 Monitor

Para ativar o monitor é necessário que o gerenciador GOD esteja sendo executado. Caso negativo devemos ativá-lo. Para tal teríamos "$maquina \$TOM_RULES\GOD\god_startup< CR > ", onde "\$TOM_RULES\GOD" é o diretório onde foi instalado o gerenciador GOD. O monitor TOM_RULES será ativado pelo comando "$maquina \$TOM_RULES\TOM_STARTUP gerenciador< CR > " onde "$TOM_RULES" é o diretório onde foi instalado o monitor, "TOM_STARTUP" é a chamada ao programa executável do monitor, e "gerenciador" o parâmetro a ser passado ao mesmo. O monitor executará em

"background" monitorando a ocorrência de eventos temporais e eventos de banco de dados. Quando detectado o evento, o monitor verificará se o mesmo foi definido como componente de alguma regra, e caso positivo, as regras correspondentes serão encarregadas de dispararem os gatilhos que as compõem.

A detecção (sinalização) de eventos se fará através de duas formas distintas: gerenciamento da estrutura AGENDA e através da comunicação com qualquer processo que esteja utilizando uma classe monitorada pelo sistema. Essas duas formas serão explicadas na próxima seção. Quando detectado um evento, todas as regras que se fizerem constituídas pelos mesmos serão imediatamente sinalizadas através do atributo *of_rule* discutido no capítulo 4, característico de todo evento. Quando um evento definido como componente de uma regra ocorre, e estando o mesmo ativo, o monitor avaliará se todos os componentes evento da regra foram sinalizados. Caso positivo a regra envia a mensagem *fire* para todos os gatilhos que compoêm seu atributo *gatilhos* obedecendo a ordem de prioridade definida nos gatilhos.

Foram definidos três tipos de mensagens que disparam gatilhos, *fire, fire_ri* e *fire_t*. A mensagem *fire* é enviada por regras que possuem um evento de banco de dados como componente *eventos*. Ela normalmente passa como parâmetro o identificador do objeto que gerou a sinalização do evento iniciador de todo o processo. A mensagem *fire_ri* passa como parâmetro um objeto descendente da classe Sather OB [OMHU91] que será explicado na próxima seção. É enviado por regras que funcionam como restrições de integridade.

A mensagem *fire_t*, disparo do gatilho sem parâmetros, é enviada por regras compostas por eventos temporais como componente do atributo *eventos*. No disparo de um gatilho a primeira avaliação a ser feita será sobre seu *status*. Caso o gatilho esteja habilitado será avaliado o parâmetro recebido (caso a mensagem seja *fire*), verificando se o mesmo faz parte do conjunto de identificadores especificados no atributo *desabilitado_para*. Caso positivo a *condição* do gatilho não será avaliada. Caso contrário será feita a avaliação da *condição*. A *condição* consiste de um conjunto de predicados ligados por conjunções e/ou disjunções.

Cada predicado constitui-se de dois operandos e um operador. Serão permitidas comparações entre atributos de um objeto, atributos de classe, parâmetros (utilizados quando os gatilhos funcionam como restrição de integridade) e constantes. Estes operandos podem pertencer a qualquer classe Sather. Os operadores são seis: *igual, diferente, maior, menor, maior* ou *igual* e *menor* ou *igual*. Em predicados construídos com objetos complexos só poderão ser utilizados os operadores relacionais *igual e diferente*. A condição retorna verdadeiro ou falso. A seguir é feita uma avaliação do parâmetro *sequência de execução*. Caso o mesmo seja igual a *equal* o sistema executará a ação componente do atributo *ação* ou *f_ação* dependendo do valor retornado pela avaliação da condição. Este modo de execução liberará o monitor para executar outras tarefas, e caso o processo de avaliação tenha sido iniciado por um evento

de banco de dados, o monitor liberará o término de execução do método inicializador. Caso a sequência de execução seja *before* o sistema bloqueará o método que ativou a regra até que seja finalizada a execução da *ação/f_ação* do gatilho em questão.

Especificar a sequência de execução como *before* é o meio de modelar restrições de integridade, já que a execução do gatilho precederá a execução do método gerador. Não é permitida a construção de regras compostas por eventos temporais e um gatilho que possua sequência de execução *before*, visto que não existe nenhuma funcionalidade nesta combinação.

O atributo *ativo* dos evento e *status* dos gatilhos tem funções similares. Mantivemos este tratamento objetivando tornar mais seletiva a inibição/desinibição dos aspectos dinâmicos.

## Comunicação

Quando um usuário executa seu programa e no mesmo é aplicado um método sobre uma classe, operação esta definida no TOM Rules como um evento de banco de dados, será enviada uma mensagem para o monitor informando a ocorrência. Isto é totalmente transparente para o usuário. Este envio de mensagens será feito através de *filas de mensagens*, que terão seu conteúdo periodicamente averiguado pelo monitor.

A necessidade deste meio de comunicação discutido anteriormente deveu-se a falta de um servidor de classe e objetos aglutinados em um mesmo sistema. Pelo mesmo procedimento são monitoradas todas as inserções, alterações e exclusões de objetos requisitadas ao GOD, de modo que são do conhecimento do monitor todos os identificadores de objetos armazenados no GOD.

## Regras

Na extensão do sistema O2 [BAUZ 91] regras comuns foram implementadas como pós-condições dos métodos enquanto as restrições de integridade foram implementadas como pré-condições dos mesmos. Aventou-se a hipótese de adotar-se uma abordagem semelhante, mas devido as limitações técnicas do gerenciador de objetos optamos por uma alteração física dos métodos. Apesar de ser dentre as alternativas disponíveis a mais eficiente, esta abordagem constitui-se no maior fator de restrição do sistema, no que diz respeito a modelagens de restrições de integridade. Como as restrições de integridade caracterizam-se por serem executadas antes do método definido como evento, a monitoração é feita

sobre os parâmetros deste método, de modo que fomos obrigados a definir uma padronização restritiva para que a monitoração fosse efetiva. Em se construindo um evento de banco de dados que funcione como restrição de integridade será necessário seguir duas normas:

- o parâmetro sujeito a monitoração deve ser identificado como *"parâmetro"* e seu tipo terá que pertencer a classe Sather "OB" ou qualquer uma de suas subclasses.
- Sather oferece tipos comuns como inteiros, caracteres, strings, reais, doubles e oferece a extensão destes tipos, as classes INT_OB, CHAR_OB, STRING_OB, REAL_OB, DOUBLE_OB, todos descendentes da classe inicial Sather "OB", com identificadores de classes e objeto. Por exemplo: um INT_OB é um objeto, com identificador de classe e objeto, que descende da classe "OB" e que funcionalmente é similar a um inteiro. Esta norma facilita o trabalho do monitor já que todo parâmetro será tratado como objeto. Portanto constitui-se mais de uma limitação declarativa do que funcional.

## Agenda

O monitor TOM Rules além de detectar os chamados eventos de banco de dados, reconhece automaticamente a ocorrência de eventos temporais. Este monitoramento basea-se no gerenciamento dinâmico de uma agenda (Figura 5.3.1), estrutura onde será armazenada uma lista cronologicamente organizada descrevendo um subconjunto de futuras ocorrências de eventos temporais. Esta abordagem é uma extensão do estudo feito em [LING87].

## Estrutura da Agenda

Uma explanação a respeito do domínio de ocorrência se faz necessária. Eventos temporais ocorrem quando seus tempos de ocorrência são iguais ao *TempoCorrente* (TempoCorrente é o valor retornado pela função *time* que lê o clock do computador), isto é, seu predicado torna-se verdadeiro para o *TempoCorrente*. Com o objetivo de prover uma maior eficiência na detecção de eventos temporais, trabalha-se na prática com intervalos de tempo, visto que em algum instante durante o intervalo de tempo seu predicado será verdadeiro. Portanto a noção de **domínio_de_ocorrência** deve ser distingüida de **tempo_de_ocorrência**

O **domínio_de_ocorrência** de um evento temporal é o intervalo de tempo durante o qual certamente o evento ocorrerá, e **tempo_de_ocorrência** o intervalo de tempo delimitado pelos atributos

*when* e *delay* de um evento. Por exemplo: se considerarmos o **domínio_de_ocorrência** < 93,03,24,10,00,00 - 93,03,24,10,00,30> e o **tempo_de_ocorrência** do evento E1 = < 93,03,24,10,00,05 - 93,03,24,10,00,20> , certamente em algum momento dentro do **domínio_de_ocorrência**o **tempo_de_ocorrência** do evento E1 será verdadeiro.

**Tempo_de_ocorrência** é o tempo preciso no qual um evento é efetivamente verificado. A ocorrência de um evento é instantânea e a asserção "**tempo_de_ocorrência** ocorre durante o **domínio_de_ocorrência**" tem que ser sempre verdadeira.

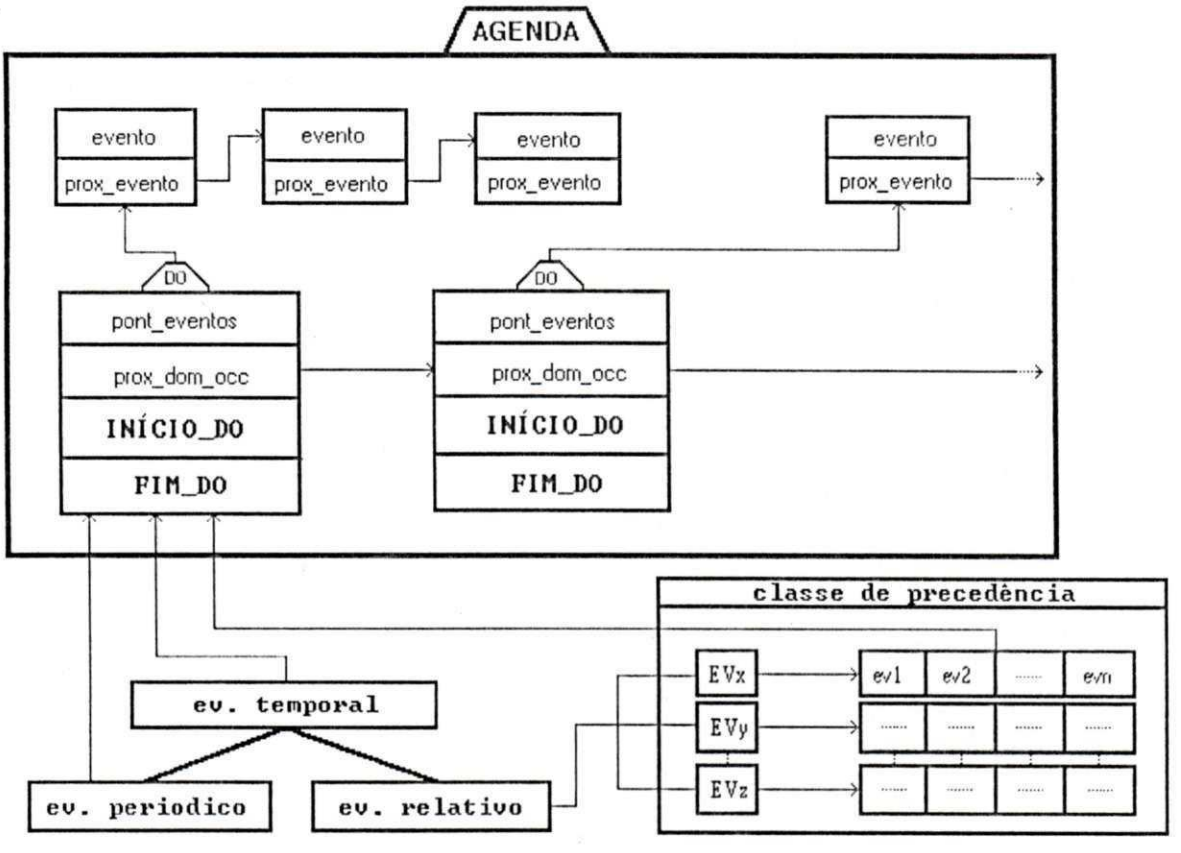


**Figura 5.3.1 Estrutura da Agenda**

## Organização da Agenda

Cada elemento da lista de domínios corresponde a um **domínio_de_ocorrência** particular e está associado a um conjunto de eventos que tem **tempos_de_ocorrência** necessariamente limitados por este domínio.

**Domínios_de_ocorrência** são definidos em termos de intervalos de tempo com [INICIO_DO - FIM_DO]. Portanto é possível também representar:

- pontos absolutos no tempo, reduzindo o **domínio_de_ocorrência** a um ponto simples (INICIO_DO = FIM_DO)
- **domínio_de_ocorrência** *infinito no futuro* através da especificando de FIM_DO como INDETERMINADO.

Dentro de cada **domínio_de_ocorrência** o conjunto de eventos é classificado em ordem crescente, de acordo com o valor de seu atributo *delay*. Isto é, como todos os eventos tem seu **tempo_de_ocorrência** determinados por um intervalo de tempo, dentro de um **domínio_de_ocorrência** os eventos que possuirem o ponto final do intervalo menor, isto é, o atributo *delay* (em termos de tempo absoluto) menor, terão maior prioridade durante a ordenação. Esta abordagem objetiva minimizar as probabilidades de não detecção de eventos temporais, já que serão priorizados dentro do **domínio_de_ocorrência** os eventos que "deixarão de acontecer" primeiro, aumentando o tempo disponível para que o monitor detecte o evento.

Os eventos são inseridos na Agenda de acordo com o seguinte algoritmo:

```
dom = agenda.domínio_de_ocorrência
if evento.delay > dom.início then
    if evento.when < dom.início then
        if evento.delay < dom.fim then                (3)
            dom <- inserir_evento(evento)
            dommodificar_início_fim_do(evento.when,evento.delay)
            agenda <- verifica_englobamento(dom)
        elsif evento.delay = dom.fim then             (2)
            dom <- inserir_evento(evento)
            dom <- modificar_início_do(evento.when)
        elsif evento.delay = dom.fim then             (1)
            agenda < - inserir_do(dom anterior,event.when,evento.delay)
        end
    elsif evento.when = dom.início then
        if evento.delay > dom.fim then                (6)
            agenda < - inserir_do(dom,event.when,evento.delay)
            agenda <- verifica_englobamento(new_do)
        elsif evento.delay = dom.fim then             (5)
            dom <- inserir_evento(evento)
        elsif evento.delay < dom.fim then             (4)
            agenda < - inserir_do(dom < - anterior,event.when,evento.delay)
        end
```

```
        elsif evento.when > dom.início then
                if evento.when = dom.fim then                    (10,11)
                        agenda < - inserir_do(dom - anterior,event.when,evento.delay)
                elsif evento.delay > dom.fim then                (9)
                        agenda <- inserir_do(dom,event.when,evento.delay)
                elsif evento.delay <= dom.fim then               (7,8)
                        dom <- inserir_evento(evento)
                end
        end
else                                                             (12)
        agenda < - inserir_do(dom,event.when,evento.delay)
end
```

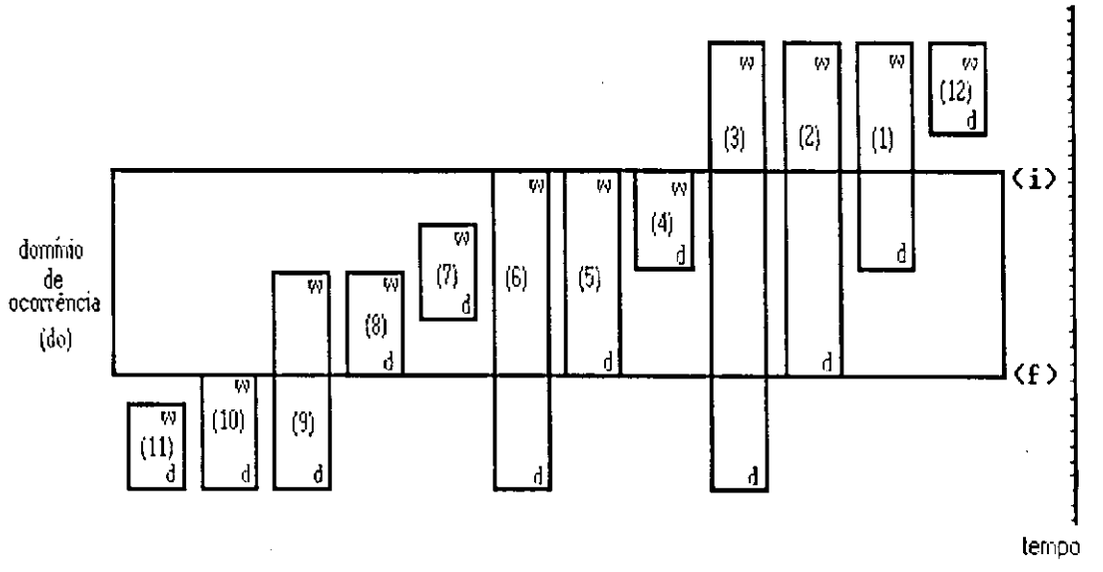


**Figura 5.3.2 Algoritmo de Inserção**

Conforme a Figura 5.3.2 consideramos a abscissa vertical uma função em relação ao tempo e o retângulo de base maior que a altura o **domínio_de_ocorrência** posicionado em função da linha de tempo (atributos início (i) e fim (f)). Os retângulos numerados são eventos temporais posicionados de acordo com seus atributos when (w) e delay (d) em relação ao **domínio_de_ocorrência.** A numeração indica no algorítmo qual o procedimento a ser tomado. Digamos que o sistema vá armazenar um evento que tenha o **tempo_de_ocorrência** na situação 2 em relação a um **domínio_de_ocorrência** (o sistema pesquisará qual dentre os **domínios_de_ocorrência** é mais adequado para se aplicar o algoritmo), isto é, seja **et** o evento temporal e **dom** o **domínio_de_ocorrência.** A situação 2 implica que:

**et.delay** > **dom.início** *and*
**et.when** < **dom.início** *and*
**et.delay** = **dom.fim**

O evento **et** será inserido no domínio_de_ocorrência **dom**. O início do *domínio_de_ocorrência* **dom (dom.início)** receberá o valor de **et.when**.

Quando iniciar-se a execução do monitor, será feita uma pesquisa sobre todos os eventos do sistema e armazenar-se-ão os eventos temporais cujos tempos_de_ocorrência forem maiores do que o *TempoCorrente*. Todos os eventos serão armazenados na agenda como se fossem intervalos absolutos no tempo. Portanto *eventos temporais, eventos temporais periódicos e eventos temporais relativos* serão armazenados de uma mesma forma na agenda.

Os *eventos temporais periódicos* terão armazenados sua próxima ocorrência, e a cada vez que o evento é detectado ele será reinserido na agenda.

Os *eventos temporais relativos* não serão inicialmente armazenados na agenda, visto ser impossível no momento de sua especificação determinar seu **tempo_de_ocorrência.** A determinação do **tempo_de_ocorrência** e a posterior inserção na agenda se fará em tempo de execução, no instante em que for detectado o evento ao qual sua ocorrência é relativa. Caso seja definido um evento temporal relativo tal que:

```
classe  EVENTO_TEMPORAL_RELATIVO
   atributos
      nome : "er_1"
      of_rule : {}
      ativo : true
      when  : <00,00,00,00,00,03>
      delay  : <00,00,00,00,00,01>
      evento_relativo : "promote_employee"
      tipo : discreto
end
```

No momento de sua especificação não será possivel a inserção na *agenda*. Caso o evento *promote_employee* seja detectado no tempo < 93,03,10,20,22,10> , será possível derivar-se um intervalo absoluto para *er_1* (transformando-o em um evento temporal absoluto), que será inserido na agenda com o **tempo_de_ocorrência** <93,03,10,20,22,13 - 93,03,10,20,22,14>.

Para otimizar a pesquisa de eventos relativos criou-se uma estrutura de **classe de precedência,** que agrupa *eventos temporais relativos* pelos eventos aos quais eles têm o tempo de ocorrência atrelado. Digamos que foram especificados os *eventos temporais relativos er_4, er_5, er_6*, tal que *er_4.evento_relativo = e_10, er_5.evento_relativo = e_11 e er_6.evento_relativo = e_10*. A estrutura de **classe_de_precedência** armazenará estas informações da seguinte maneira:

**Class_prec**(*e_10*) = {*er_4,er_6*}

**Class_prec**(*e_11*) = {*er_5*}

Quando um evento **ev** ocorre, o monitor verificará se o mesmo é chave de alguma **classe_de_precedência**, e caso positivo derivará os tempos absolutos de ocorrência dos *eventos temporais relativos* a **ev**, e os armazenará na *agenda*.

Somente futuros **domínios_de_ocorrência** estão presentes na *agenda*. O primeiro elemento da *lista de domínios* é sempre o próximo domínio a ocorrer. Portanto o monitor tem que esperar que o *TempoCorrente* esteja dentro do primeiro **domínio_de_ocorrência**. Caso positivo o monitor verificará a partir do primeiro evento se o *TempoCorrente* está inserido no **tempo_de_ocorrência** do mesmo e caso positivo ele será sinalizado.

## 5.4 Interface

Objetivando suprir ao usuário um ambiente completo para especificação de aspectos dinâmicos, o sistema provê uma interface amigável que torna simples e rápida a construção e consulta de eventos, gatilhos e regras.

## 5.4.1 Funções da Interface

Nesta seção serão descritas as funções da interface, mostrando como se comportarão as informações em dois tipos de diálogos escolhidos.

### Apresentar Informações

Foram utilizados dois meios de apresentar informações:

- *Browse* - O sistema mantem listas de regras, eventos, gatilhos, ações e classes.
- *Informação Incompleta* - Permite que a informação que já se encontra na tela faça parte da entrada do usuário, o que acelera a interação e evita erros. Utilizada na introdução de atributos de eventos e gatilhos.

### Selecionar Opções

Quando houver recursos para a manipulação direta da informação, esta se fará através de um *"click"* de *mouse*. A confirmação ou não de operações será feita através de um *"click"* de *mouse* sobre os botões de confirmação e cancelamento respectivamente.

### Fornecer Ajuda On-Line

Para dar suporte a construção de regras, eventos e gatilhos, está disponível um sistema de ajuda sensível ao contexto, ativado através da tecla < F1 >.

### Executar Informações

As operações sobre informações disponíveis nas janela são:

- criação, alteração e remoção de *regras, eventos de banco de dados, eventos temporais, eventos temporais absolutos, eventos temporais relativos e gatilhos*
- inserção e remoção de *ações*
- atualização automática do monitor

### Emitir Saída das Informações

Através de *"click"* de *mouse* sobre o *browse* de aspectos dinâmicos o sistema proverá meios de consultar *regras, eventos, gatilhos* e arquivos fontes das *ações*.

### Detecção de Erros

Mensagens de erro serão apresentadas quando procedimentos errôneos forem detectados. As mensagens serão apresentadas nas áreas de ocorrência do erro.

### Manipulação de Janelas

Todas as janelas são manipuladas através de funções do gerenciador de janelas GUIDE. Destacamos as seguintes funções:

- seleção, movimentação e remoção de janelas.
- "scroll bar" dentro de uma janela através da movimentação do *mouse*.
- transformação de janela em ícone (função oferecida apenas para a janela principal).

## 5.4.2 Características dos Usuários

O usuário desta aplicação necessita ter conhecimentos teóricos a respeito de orientação a objetos e conceitos de aspectos dinâmicos. É essencial um conhecimento prévio dos propósitos da aplicação e seria interessante conhecimentos sobre o ambiente OPENWINDOWS.

Para tornar mais simples a interação com o sistema, o mesmo é dotado de uma ajuda *"on-line"*, onde o usuário obtém informações imediatas, de acordo com o contexto onde ele se encontrar.

## 5.4.3 Descrição da Interface

### Inicialização do Sistema

Para inicializar a interface do sistema devem ser executados os seguintes procedimentos:

. verificar se o gerenciador de objetos GOD está executando. Caso o gerenciador não esteja executando, será necessário ativá-lo.

. ativar a interface através do comando "$maquina \$TOM_RULES\ TOM_STARTUP interface<CR>" onde "$TOM_RULES" é o diretório onde foi instalado o monitor, "TOM_STARTUP" é a chamada ao programa executável do monitor, "interface" o parâmetro a ser passado ao mesmo , e < CR> a indicação do acionamento da tecla "ENTER". Após este procedimento o usuário terá a sua disposição a janela base da interface do sistema.

### Softwares Utilizados

Na implementação da interface do sistema foram utilizadas as linguagens de programação C e Sather e a ferramenta de desenvolvimento de janelas GUIDE.

### Estilo de Diálogo

O estilo de diálogo escolhido foi a *informação incompleta*, auxiliado por "browse", por ser um estilo básico oferecido pela ferramenta de desenvolvimento de janelas GUIDE, tornado mais rápida, natural e estimulante a interação do usuário com a interface.

Informações apresentadas através do estilo "browse" permitem que o usuário tenha uma visão abrangente de todos os objetos monitorados pelo sistema, tornando mais simples a construção e verificação dos aspectos dinâmicos.

## 5.4.4 Apresentação da Informação

A interface constitui-se de nove janelas, sendo uma janela base e as restantes janelas *pop_up* ativadas a partir da janela básica. A seguir mostraremos como é apresentada a informação em cada janela.

A *janela base (Figura 5.4.4.1) divide-se em seis áreas distintas:*

- área de botões de controle
- área de eventos
- área de regras
- área de gatilhos
- área de ações
- área de linha de comando
- área de ativação de comandos UNIX

A área de botões de controle localiza-se na porção superior da *janela base* consiste de sete botões:

- *botão de inserção* - insere regras, eventos, gatilhos e ações.
- *botão de alteração* - altera regras, eventos, gatilhos.

- *botão de exclusão* - exclui regras, eventos, gatilhos e ações.
- *botão de classes* - ativa a janela *pop_up* utilizada para inserir novas classes a serem monitoradas pelo sistema.
- *botão de atualização do monitor* - torna ativas todas as alterações feitas pelo usuário através da interface.
- *botão de confirmação* - confirma opções da área de botões.
- *botão de cancelamento* - cancela opções da área de botões.

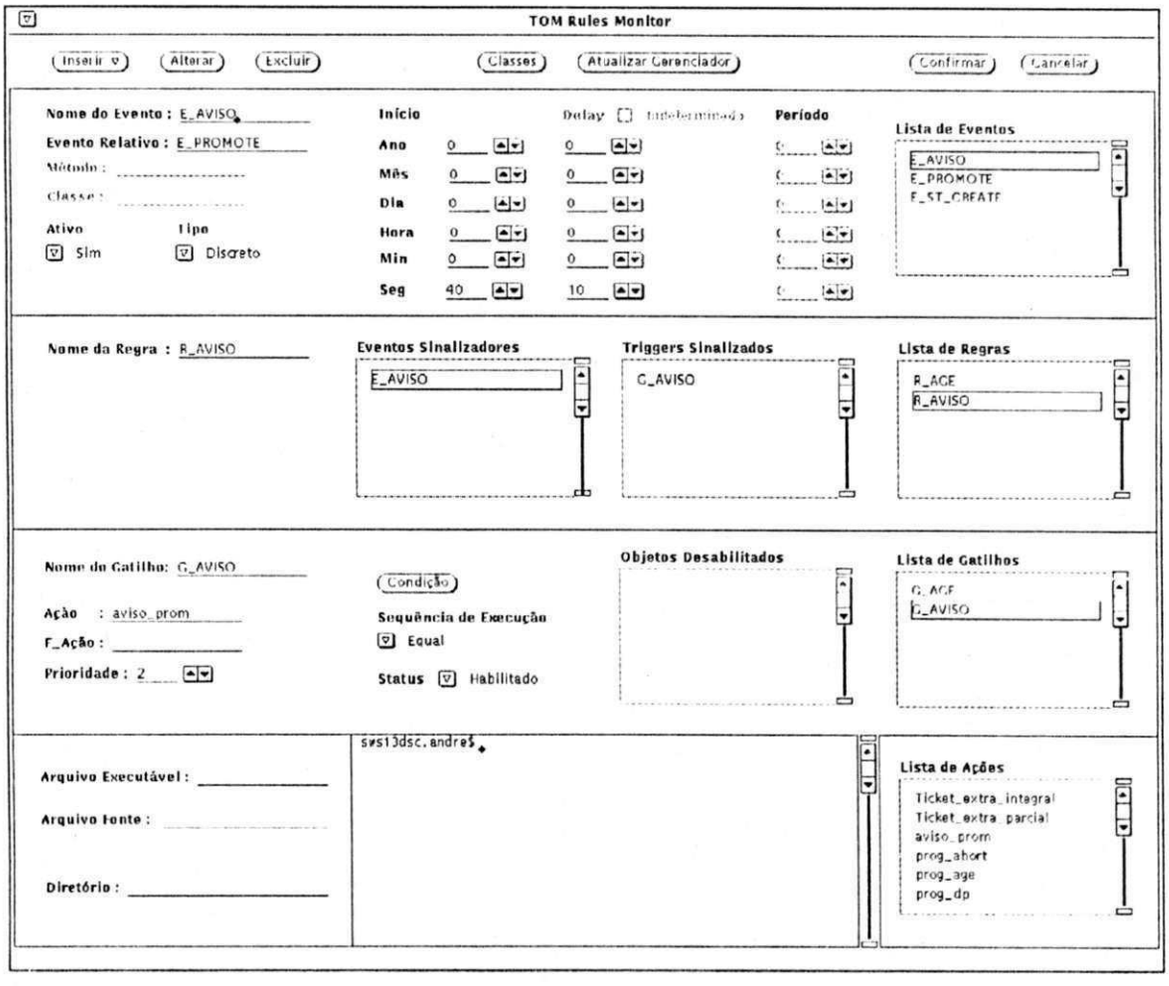


**Figura 5.4.4.1 Janela Base**

A área de eventos consiste de um *"browse"* com a lista de eventos monitorados pelo sistema e um conjunto de campos que constituem os atributos de um evento. De acordo com o tipo de evento (banco de dados, temporal, temporal relativo e temporal periódico) alguns campos serão desativados. Um *"click"* de *mouse* sobre um elemento da lista de eventos ocasionará o preenchimento e a apresentação dos atributos do evento selecionado. A diferenciação entre tipos de eventos é feita através da inibição de certos campos. A inserção, alteração e exclusão de eventos serão feitas através da interação com os botões de controle *Inserir, Alterar* e *Excluir.*

A área de regras consiste de um *"browse"* com uma lista das regras monitoradas pelo sistema, um campo de identificador de regra e listas para manipulação de gatilhos e eventos que compõem a regra. Um *"click"* de *mouse* sobre um elemento da lista de regras ocasionará o preenchimento e a apresentação dos atributos da regra selecionada. A inserção, alteração e exclusão de gatilhos serão feitas através da interação com os botões de controle *Inserir, Alterar e Excluir.*

A área de gatilhos consiste de um "*browse*" com uma lista de gatilhos monitoradas pelo sistema e um conjunto de campos que constituem os atributos de um gatilho. Um *"click"* de *mouse* sobre um elemento da lista de gatilhos ocasionará o preenchimento e a apresentação dos atributos do gatilho selecionado. A inserção, alteração e exclusão de gatilhos serão feitas através da interação com os botões de controle *Inserir, Alterar e Excluir.*

A área de ações consiste de um *"browse"* com uma lista de ações utilizadas para construção de gatilhos e um conjunto de campos utilizados para informar a localização lógica de uma ação a ser inserida no sistema. Um *"click"* de *mouse* sobre um elemento da lista ações ocasionará a apresentação do arquivo fonte da ação escolhida caso o mesmo exista. A inserção e exclusão de ações serão feitas através da interação com os botões de controle *Inserir* e *Excluir.*

Durante a manipulação das áreas de evento, regra, gatilho e ação é feita uma crítica de consistência, e em caso de erro, mensagens serão apresentadas dentro das áreas de ocorrência específicas.

A *janela de condição* (Figura 5.4.4.2) é uma janela *pop_up* ativada pelo *"click"* de *mouse* sobre o botão *condição* da área de gatilhos da janela base. É utilizada para construir, alterar e consultar a parte condicional do gatilho. Consiste-se de:

- uma lista que apresenta todas as classes monitoradas pelo sistema (quando o gatilho não funciona como restrição de integridade, uma opção tem que ser escolhida durante a criação da condição).
- uma lista de visualização dos predicados que formam a condição.
- uma lista de visualização dos operadores lógicos que formam a condição.
- dispositivo de opções de escolha de operadores lógicos (and/or). Um *"click"* de *mouse* sobre uma das opções do dispositivo inserirá uma das opções na lista de visualização dos operadores lógicos.
- botão de *predicado*. Através de um "*click*" de *mouse* ativará a janela *pop_up* responsável pela construção dos predicados. Cada construção/alteração/exclusão de predicados implicará numa atualização automátiva da lista de visualização de predicados.
- botão de confirmação - confirma operação.
- botão de cancelamento - cancela operação.

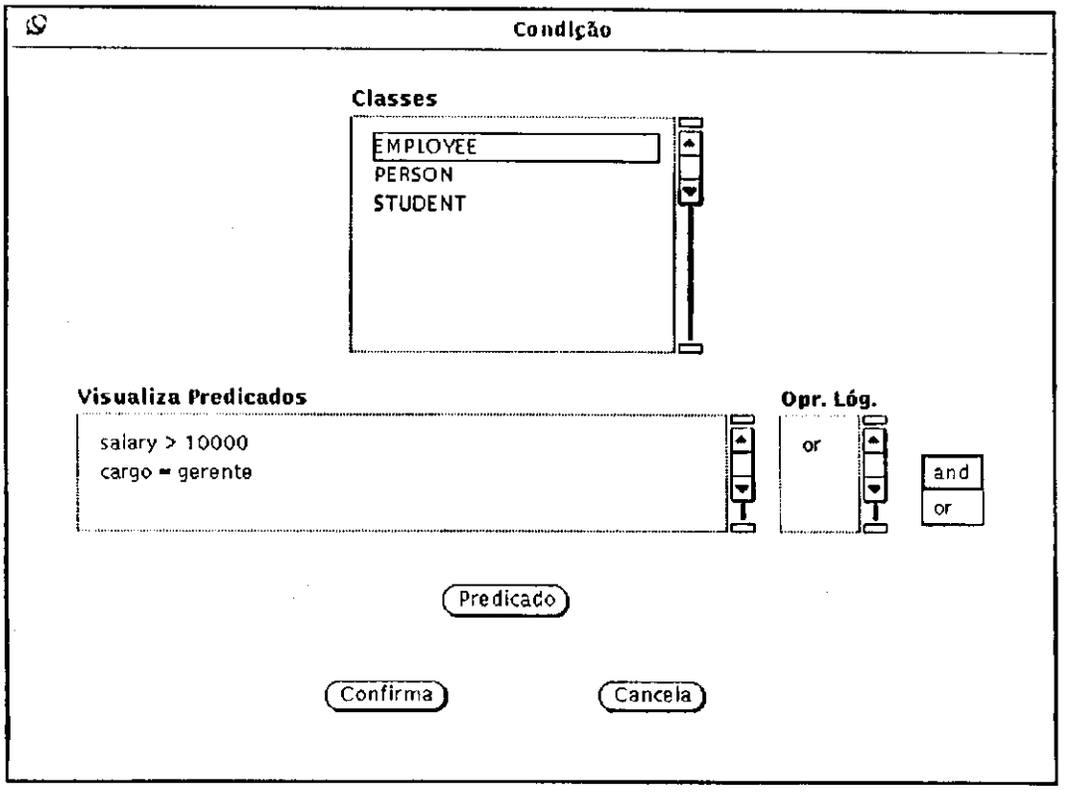


**Figura 5.4.4.2 Janela de Condição**

A escolha da sequência de execução de um gatilho *(janela base - área de gatilhos)* determina uma distribuição de seis janelas do sistema em dois grupos iguais, um referente a escolha da opção *equal* (*janelas de predicado, operando_direito e operando_esquerdo*), e outro referente a escolha da opção *before (janelas de predicado_integridade, operando_de_integridade_direito e operando_de_integridade_esquerdo).* Isto se deve a diferenciação em termos de parâmetros de execução que existe entre gatilhos que funcionam como restrição de integridade e os que não possuem esta função. Quando um gatilho funciona como restrição de integridade ele recebe como parâmetro um valor qualquer, enquanto que quando ele não tem esta função, o gatilho recebe um objeto pertencente a alguma classe do sistema. Sendo assim, os construtores de predicado são diferentes, o que exigiu a manutenção de janelas diferentes para o mesmo propósito.

As mensagens de erro nestas janelas serão apresentadas de acordo com o posicionamento do *mouse* no momento de detecção do erro.

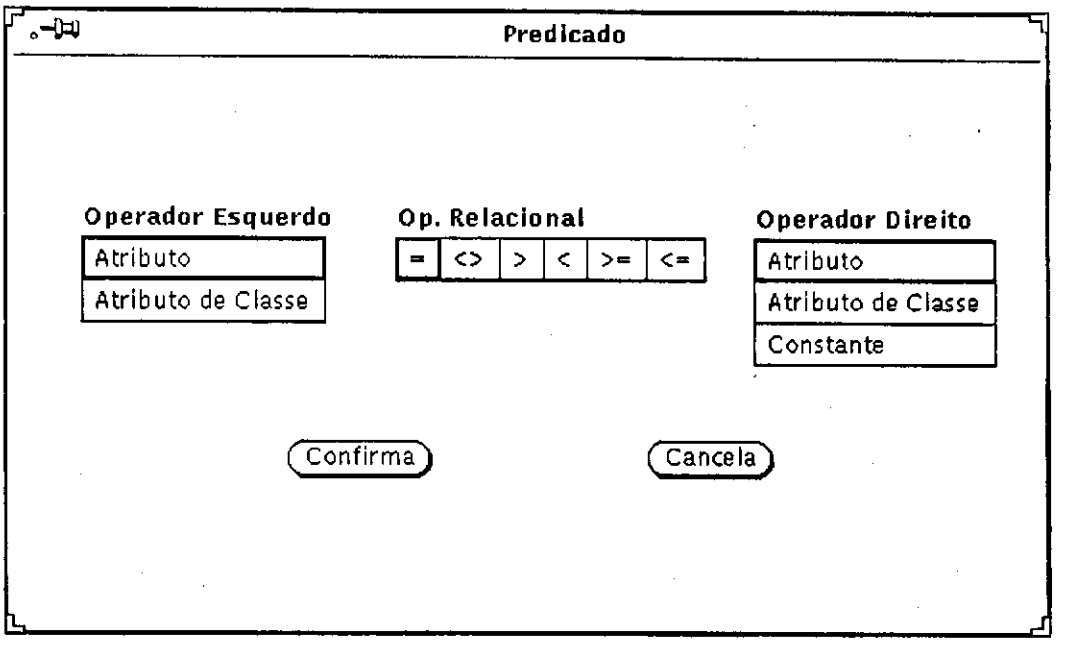


**Figura 5.4.4.3 Janela Predicado**

*A janela de predicado* (Figura 5.4.4.3) constitui-se de três conjuntos de opções: uma para escolha de operadores relacionais (=, <>, <, >, <=, >=), uma para escolha de *operando esquerdo (Atributo, Atributo de Classe)* e uma para escolha de *operando direito (Atributo, Atributo de Classe, Constante).* As duas últimas ativarão janelas *pop_up* responsáveis pela especificação das opções. Possui também botões para confirmação/cancelamento das opções.

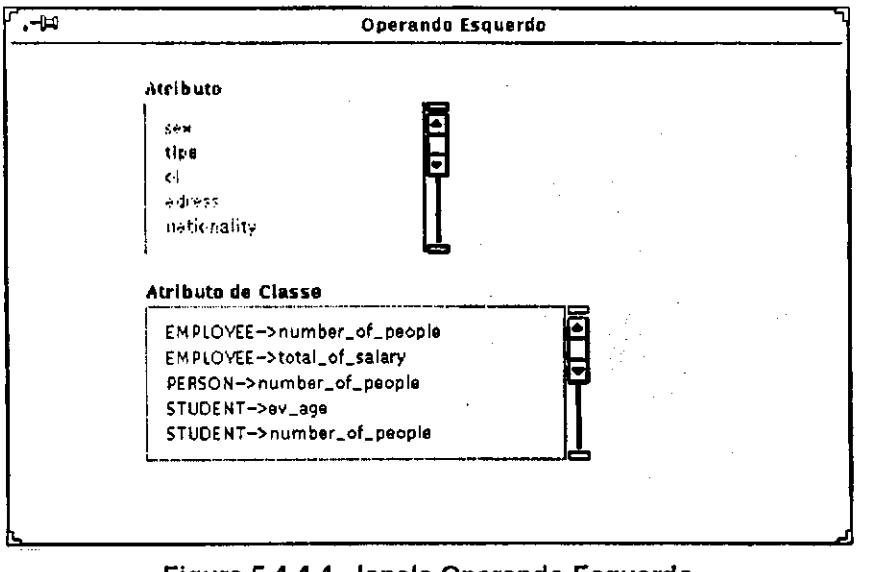


**Figura 5.4.4.4 Janela Operando Esquerdo**

Dependendo da escolha feita dentre as opções da *janela de predicado*, será mostrado na *janela de operando_esquerdo (Figura 5.4.4.4)* uma lista de atributos de uma classe (classe previamente

escolhida através de uma seleção de ítem na janela de condição) ou uma lista de atributos compartilhados das classes do sistema (atributos que possuem um mesmo valor para todas os objetos de uma determinada classe monitorada pelo sistema). Uma dentre as opções tem que ser escolhida.

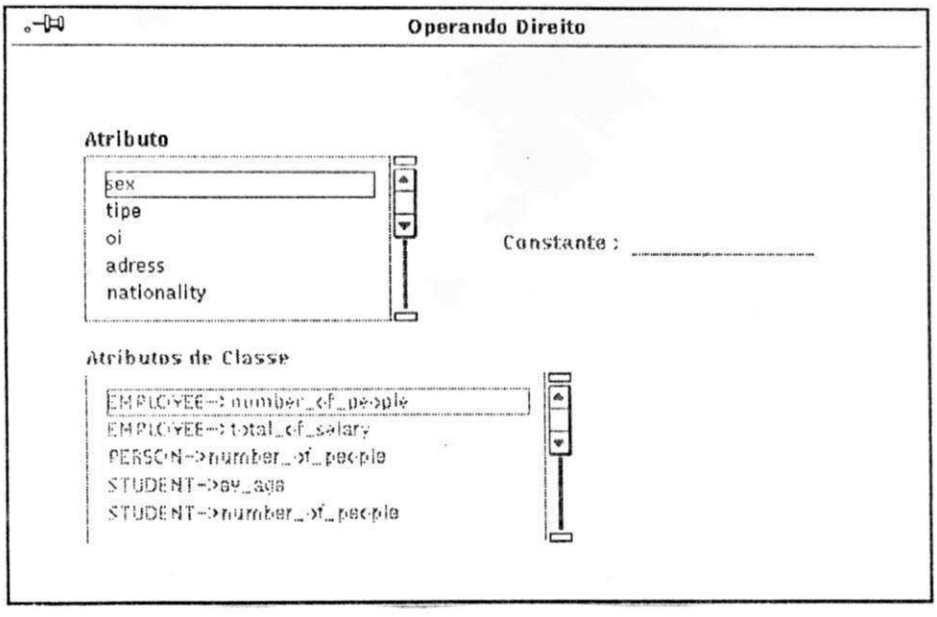


**Figura 5.4.4.5 Janela Operando Direito**

Funcionalmente similar à janela descrita anteriormente, *a janela de operando_direito* (Figura 5.4.4.5) diferencia-se pela apresentação de um campo texto para inserção de uma constante, caso seja esta a opção escolhida na janela de predicado.

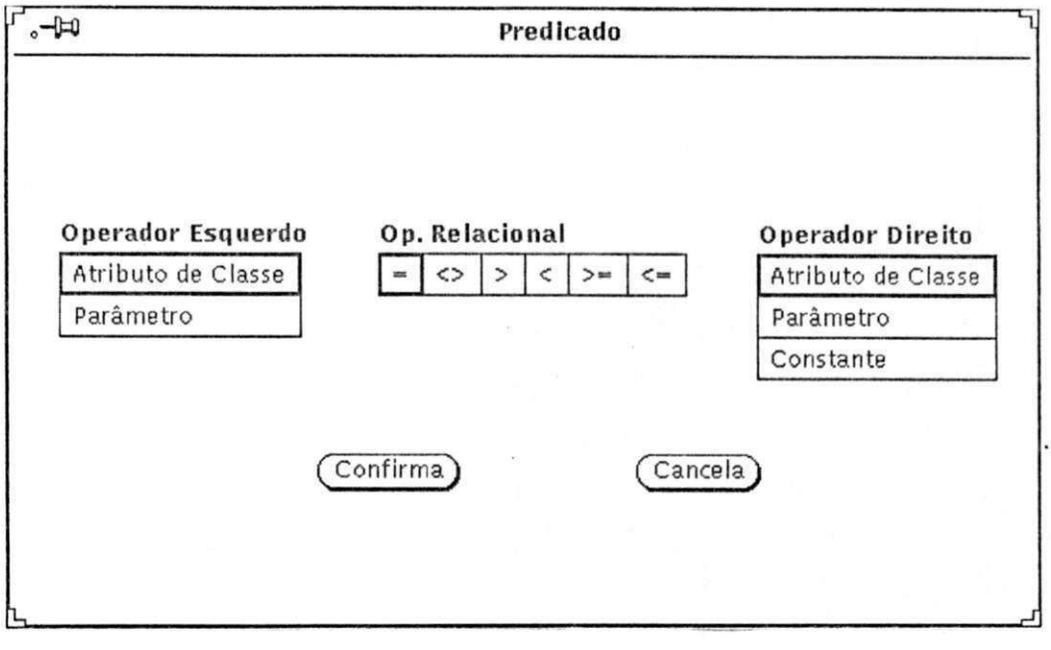


**Figura 5.4.4.6 Janela Predicado Integridade**

Similar à janela de predicado, a *janela de predicado_integridade* (Figura 5.4.4.6) diferencia-se pela não existência da opção *Atributo* e pela presença da opção *Parâmetro* nas opções de escolha de operadores.

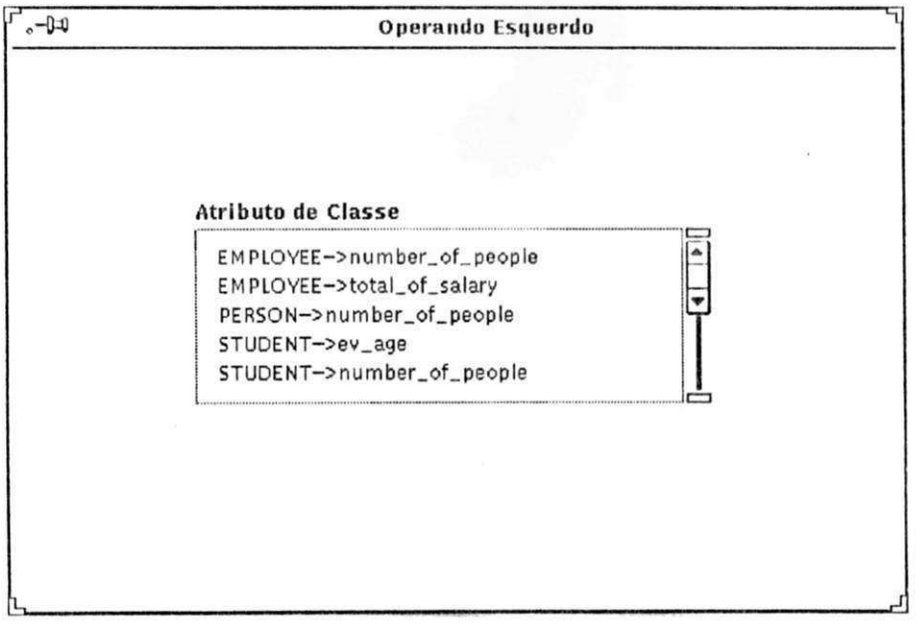


**Figura 5.4.4.7 Janela Operando_Integridade_Esquerdo**

Similar à janela de operando_esquerdo, *a janela de operando_integridade_esquerdo* (Figura 5.4.4.7) diferencia-se pala inexistência da lista de atributos.

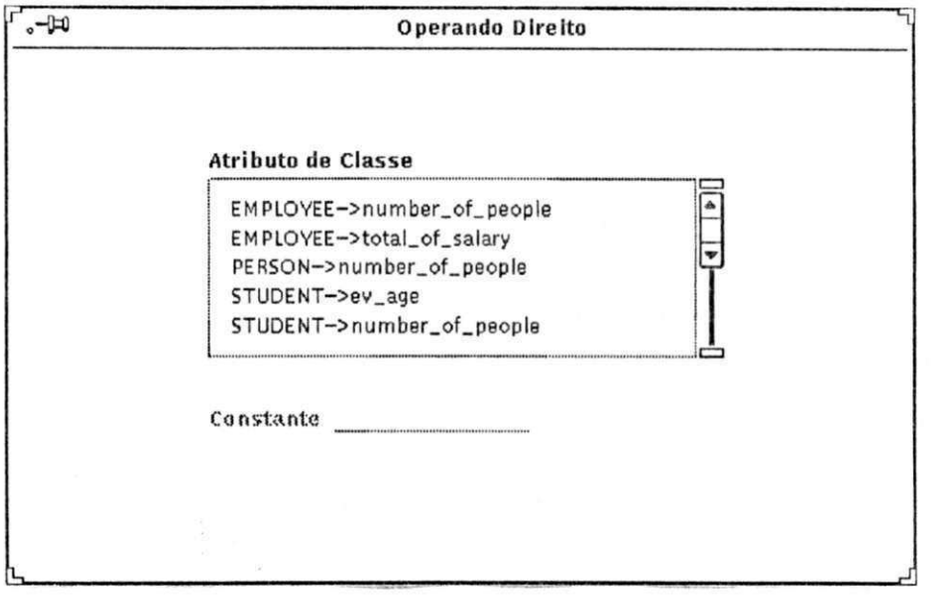


**Figura 5.4.4.8 Janela Operando_Integridade_Direito**

Similar a *janela de operando_direito*, a *janela de operando_integridade_direito (Figura 5.4.4.8)* diferencia-se pela inexistência da lista de atributos.

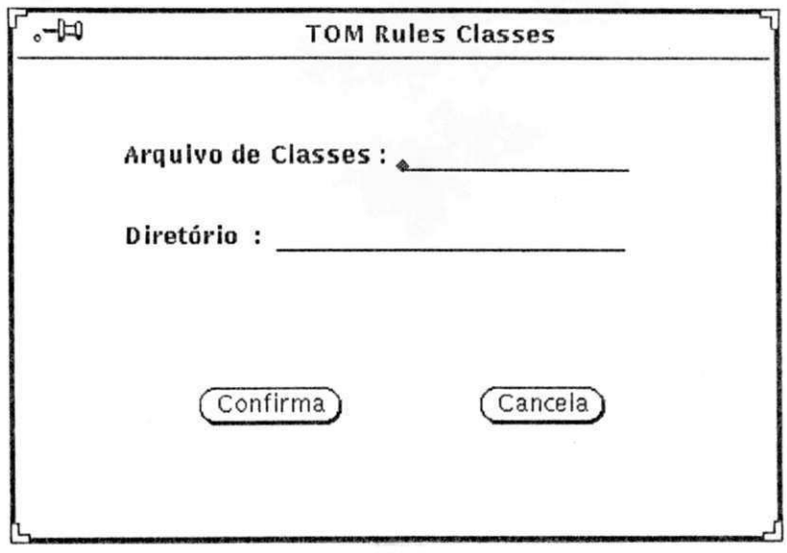


**Figura 5.4.4.9 Janela de Classes**

Ativada através do botão *Classe* na área de botões de controle da janela base. A *janela de classes* (Figura 5.4.4.9) é constituída de campos para especificação da localização e identificação dos arquivos de classe no formato Sather onde estão especificadas as novas classes a serem monitoradas pelo sistema.

## 5.4.5 Simulação de Utilização do Sistema

Após a execução dos procedimentos citados no tópico *Inicialização do Sistema*, a janela base é apresentada(Figura 5.4.5.1).

Caso se deseje alguma informação de ajuda basta que se posicione o *mouse* sobre o ítem desejado e apertar a tecla <F1>. Será mostrada uma janela subdividida em duas partes, uma apresentando a ajuda com breves explicações sobre o ítem em questão, e a outra no canto esquerdo da janela mostrando o ítem do qual se desejou maiores informações. A Figura 5.4.5.2 apresenta a janela de ajuda ativada quando o *mouse* está posicionado sobre a *lista de eventos*

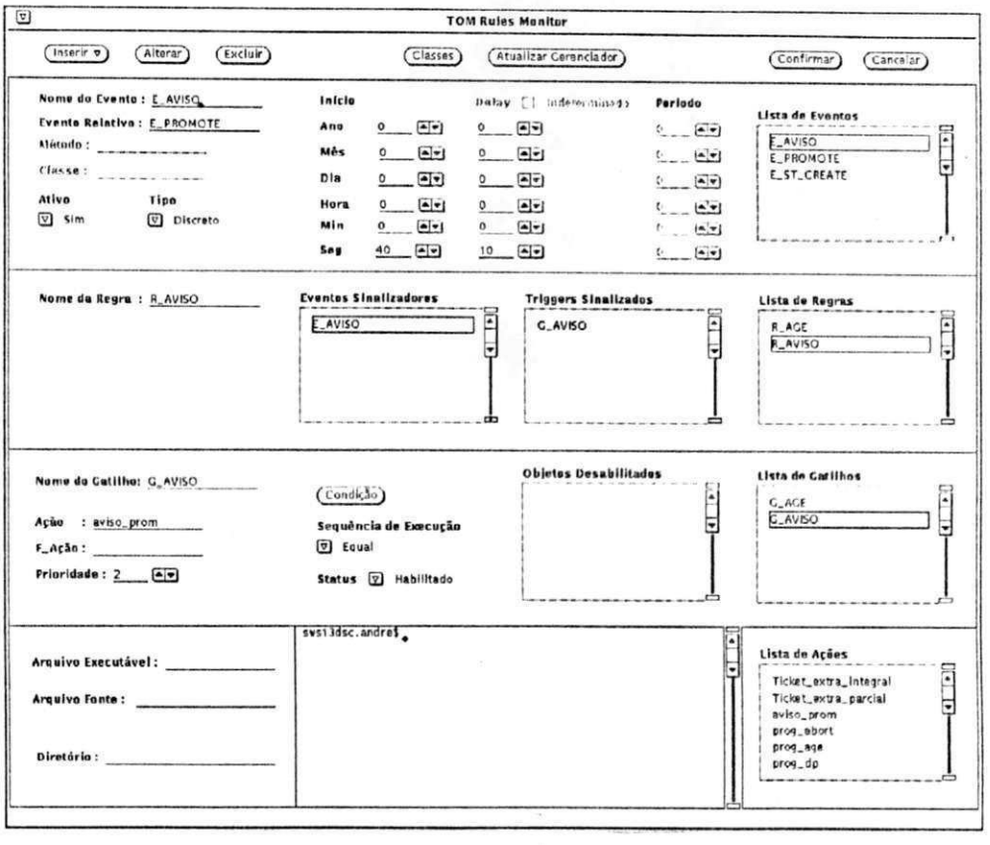


**Figura 5.4.5.1 Janela Base - Inicialização**

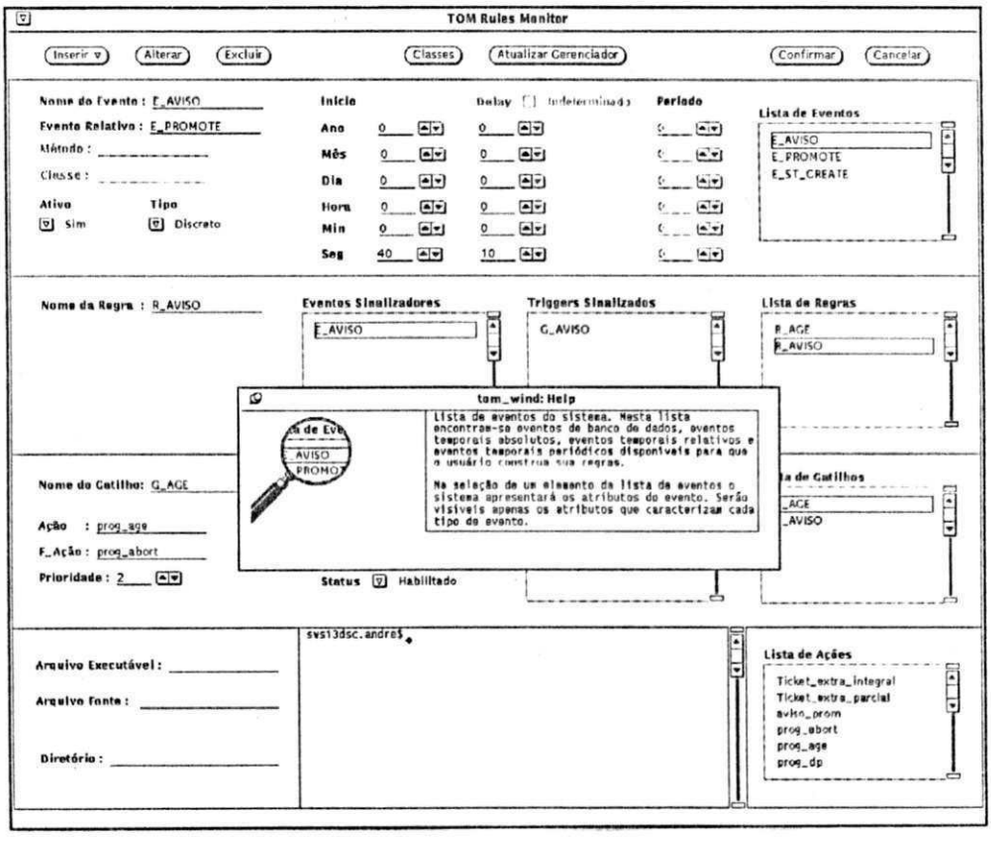


**Figura 5.4.5.2 Janela Base com help ativo**

A seguir mostraremos numa série exemplos de como manusear a interface. Basearemo-nos nos exemplos do capítulo 4. Inicialmente mostraremos quais os passos necessários para a construção do evento de banco de dados *transfere_p_matriz.* O primeiro passo é *"clickar" o mouse* sobre o botão *Inserir* na área de botões de controle. O sistema apresentará um menu de opções *(Eventos, Regras, Gatilhos, Acões).* Repetiremos o mesmo procedimento sobre a opção *Eventos.* O sistema apresentará um submenu com as opções *Evento de Banco de Dados, Evento Temporal Absoluto, Evento Temporal Periódico e Evento Temporal Relativo.* Selecionaremos a primeira opção.

Agora a área de eventos está pronta para receber as informações que constituirão o evento *transfere_p_matriz* Preencheremos o campo *Nome* com"transfere_p_matriz", *o campo Método* com "transfer_matiz", o campo *Classe* com"EMPLOYEE", e escolheremos a opção "Sim" no campo *Ativo.* A seguir selecionaremos com o *mouse* o botão *Confirma* da área de *botões de controle,* estando portanto confirmada a inserção do evento. Na Figura 5.4.5.3 a situação da janela base após a inserção de *transfere_p_matriz.*

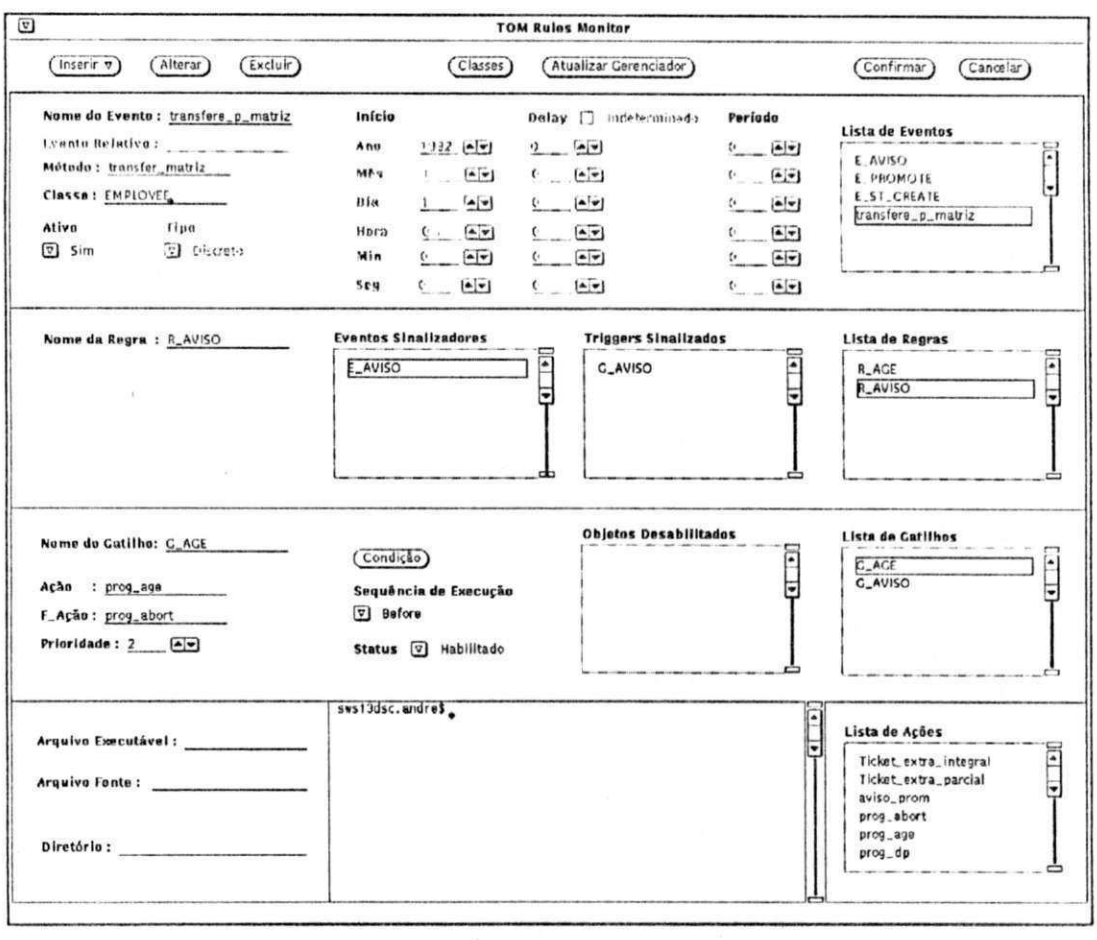


**Figura 5.4.5.3 Janela Base com transfere_p_matriz**

A seguir construiremos o evento *início_mês.* Começamos pela escolha da opção *Eventos Temporais Periódicos* do submenu do botão *Inserir.* Preencheremos o campo *Nome* com"início_mês",

escolheremos a opção "Contínuo" no campo *Tipo* e a opção "Sim" no campo *Ativo*. Preencheremos os campos de "lay_out" *Inicio* com os valores (Ano=93, Mês=01, Dia=01, Hora=00, Min=00, Seg=00), de "lay_out" *Fim* com os valores (Ano=00, Mês=00, Dia=10, Hora=00, Min=00, Seg=00) e de "lay_out" *Período* com os valores (Ano= 00, Mês= 01, Dia= 00, Hora= 00, Min= 00, Seg= 00). A seguir selecionaremos então com o *mouse* o botão *Confirma* da área de *botões de controle,* estando portanto confirmada a inserção do evento. A Figura 5.4.5.4 apresenta a janela base após a inserção de *início_mês*.

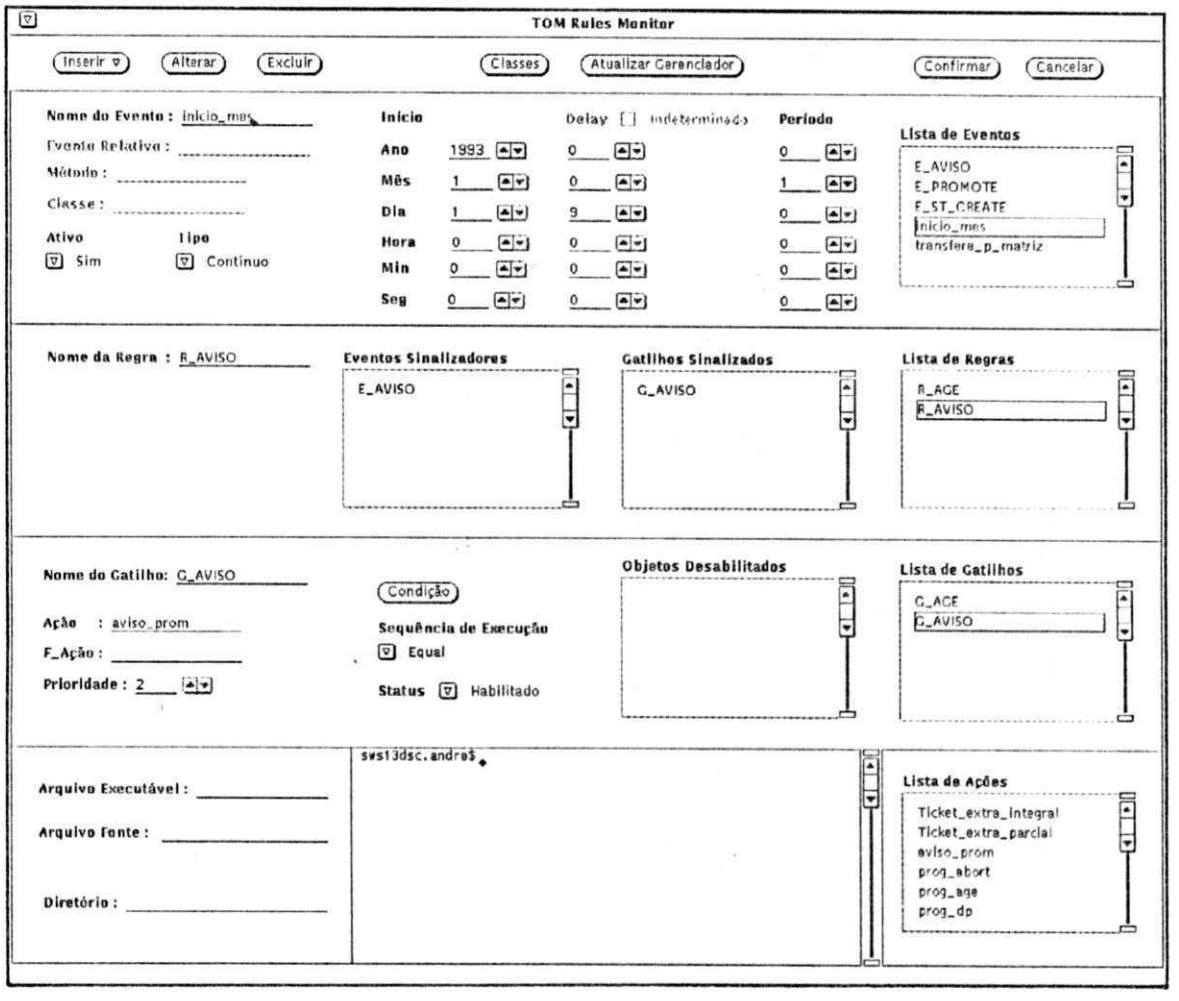


**Figura 5.4.5.4 Janela Base com inicio_mes**

A seguir construiremos o gatilho *Ticket*. Começamos pela escolha da opção *Gatilhos do menu do botão Inserir*. Preencheremos o campo *Nome* com "Ticket", escolheremos a opção "Equal" no campo *Sequência de Execução* e a opção "Habilitado" no campo *Status*. Preencheremos os campos *Ação e F_Ação* com os valores "Ticket_extra_integral" e "Ticket_extra_parcial", e o campo *Prioridade* com o valor 2. O próximo passo é especificar a *condição* do gatilho. Para isto selecionamos o botão *Condição* da

área de gatilhos. Esta operação ativará a *janela de classes*. Como escolhemos a opção "Equal" no campo *Sequência de Execução*, devemos fazer a escolha de uma dentre as classes do sistema apresentadas na lista de classes para que sejam carregados os atributos desta classe na lista de operandos. Então selecionamos o ítem "EMPLOYEE" da *lista de classes*. A seguir selecionaremos o botão *Predicado* para que seja construído o primeiro predicado da condição (self.salary < 2000). A Figura 5.4.5.5 apresentará a janela de predicados. Escolheremos a seguir o operador lógico, os operandos esquerdo e direito e o operador relacional.

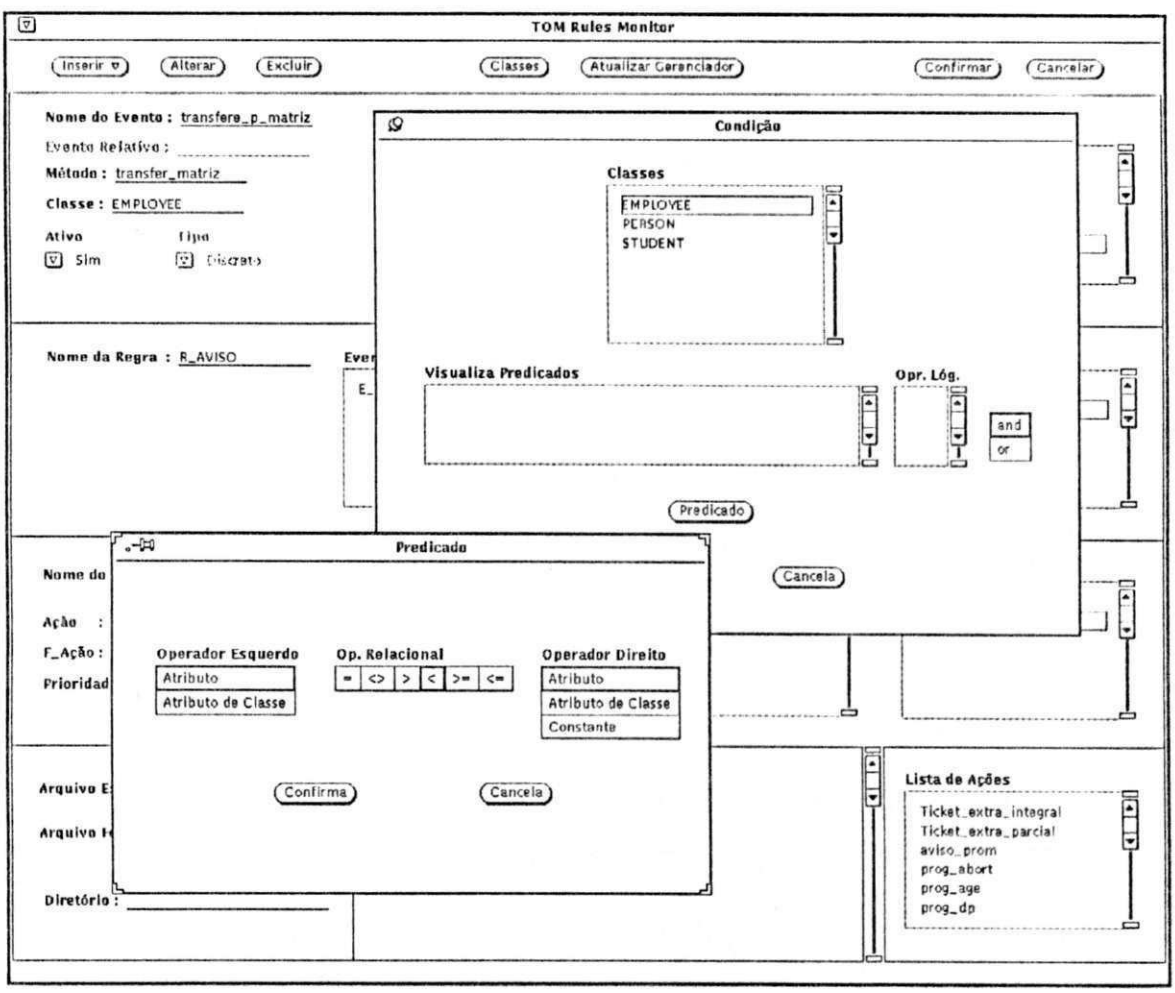


**Figura 5.4.5.5 Janela de Predicado ativada**

O operador relacional do predicado será escolhido através de um *"click" de mouse* sobre o ítem "<" do conjunto de opções de operador relacional.

O operando esquerdo será escolhido pela seleção da opção "Atributo" da janela de predicados, que ativará a janela de operando_esquerdo (Figura 5.4.6). Para complementar basta que se selecione o atributo *salário* da lista de atributos da janela.

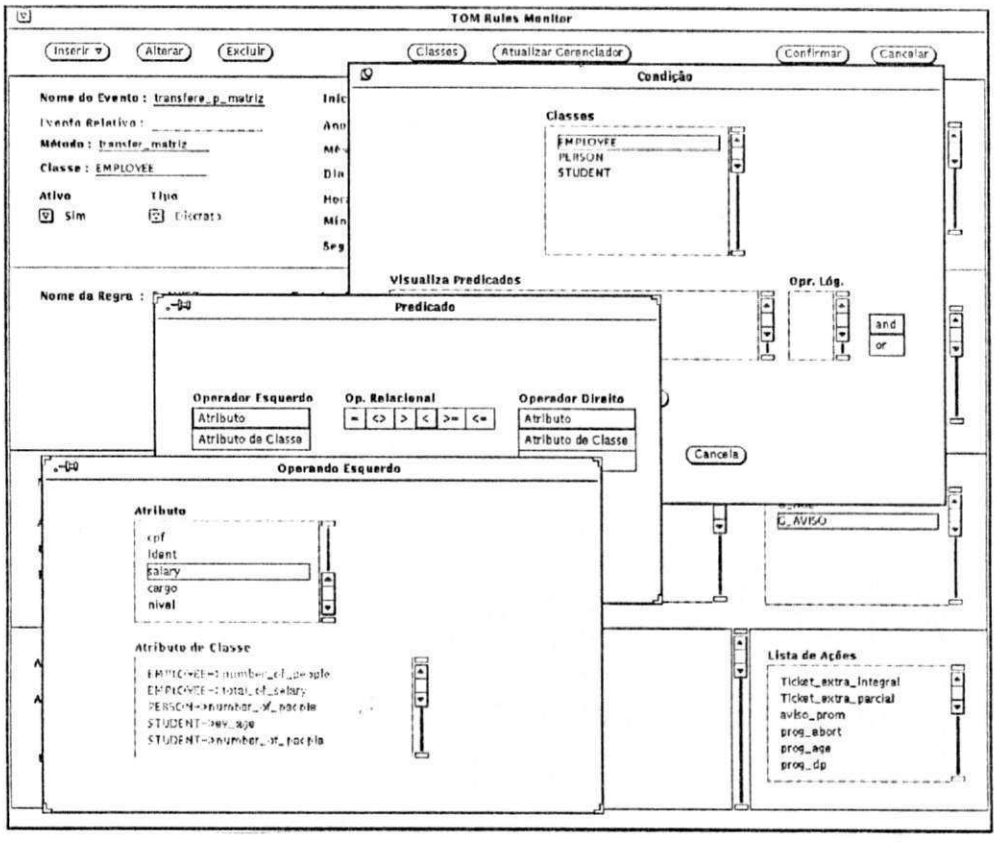


**Figura 5.4.5.6 Janela de operando_esquerdo ativa**

O operando direito será escolhido pela seleção da opção "Constante" da *janela de predicados,* que ativará a *janela de operando_direito.* (Figura 5.4.5.7) Para complementar basta que se preeencha o campo "Constante" da *janela de operando_direito* com o valor 2000.

A confirmação virá da seleção do botão *"Confirma" da janela de predicado. A janela de predicado* desaparecerá e poderá ser visto na *lista de visualização de predicados da janela de condição* o novo predicado inserido. Para inserir o segundo predicado (self.cargo = "carregador") o procedimento é similar. Para escolher o operador lógico basta que seja selecionada a opção "Or" do *dispositivo de operadores lógicos.* Para confirmar basta selecionar o botão *Confirma da janela de condição* e a mesma desaparecerá. Para finalizar bastar selecionar o botão *"Confirma" da janela base,* estando então confimada a inserção do gatilho. A janela de predicado desaparecerá. Na Figura 5.4.5.7 apresentamos a situação da janela base após a inserção de *Ticket.*

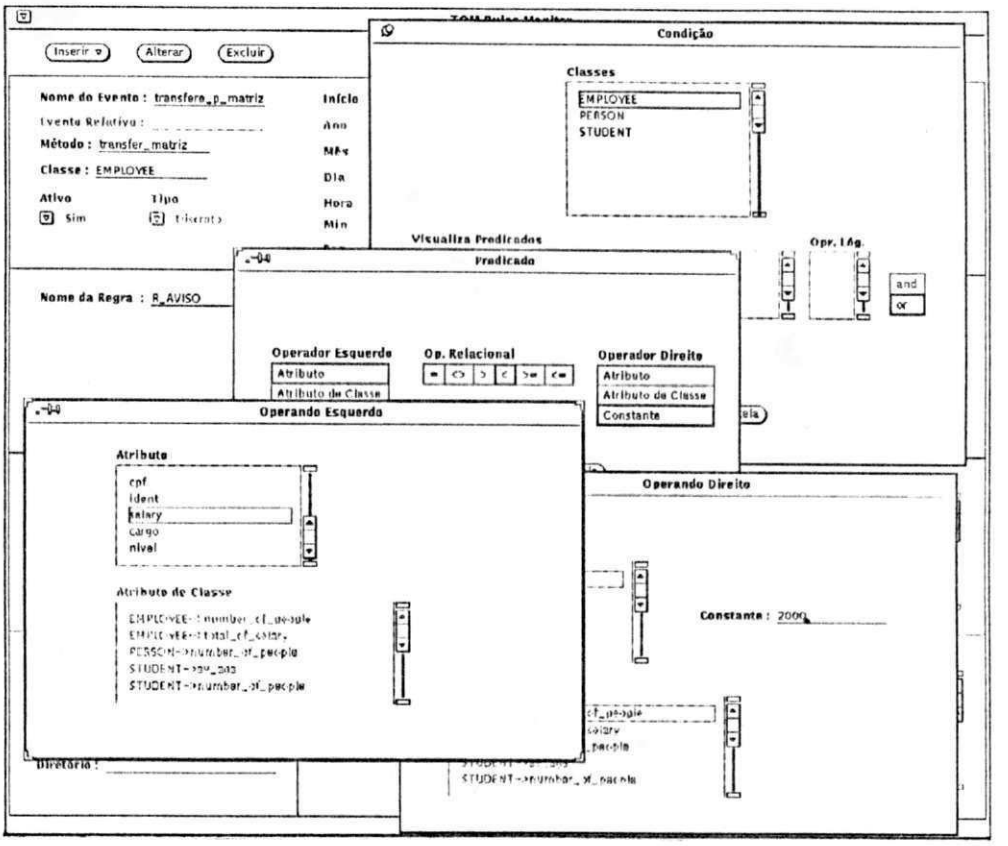


**Figura 5.4.5.7 Janela de operando_direito ativa**

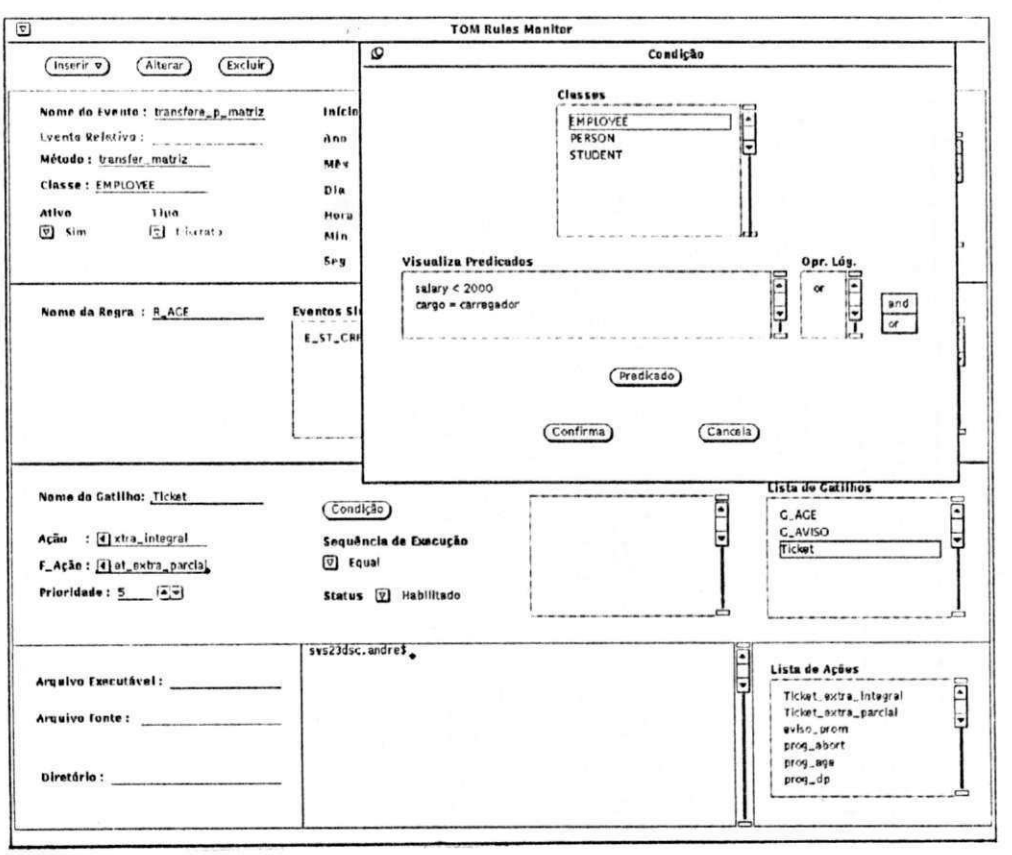


**Figura 5.4.5.8 Janela Base com Ticket**

Para construir a regra *Regra_Ticket* primeiro devemos selecionar a opção "Regras" do menu do botão *Inserir*. Preencheremos o campo *Nome* com "Regra_Ticket", a *lista de eventos* com os identificadores de evento "início_mês" e "transferir_p_matriz", e a *lista de gatilhos* com o identificador de gatilho "Ticket". Para finalizar basta selecionar o botão *Confirma* da janela base, estando então confimada a inserção do regra. Na Figura 5.4.5.9 a situação da janela base após a inserção de *Regra_Ticket*

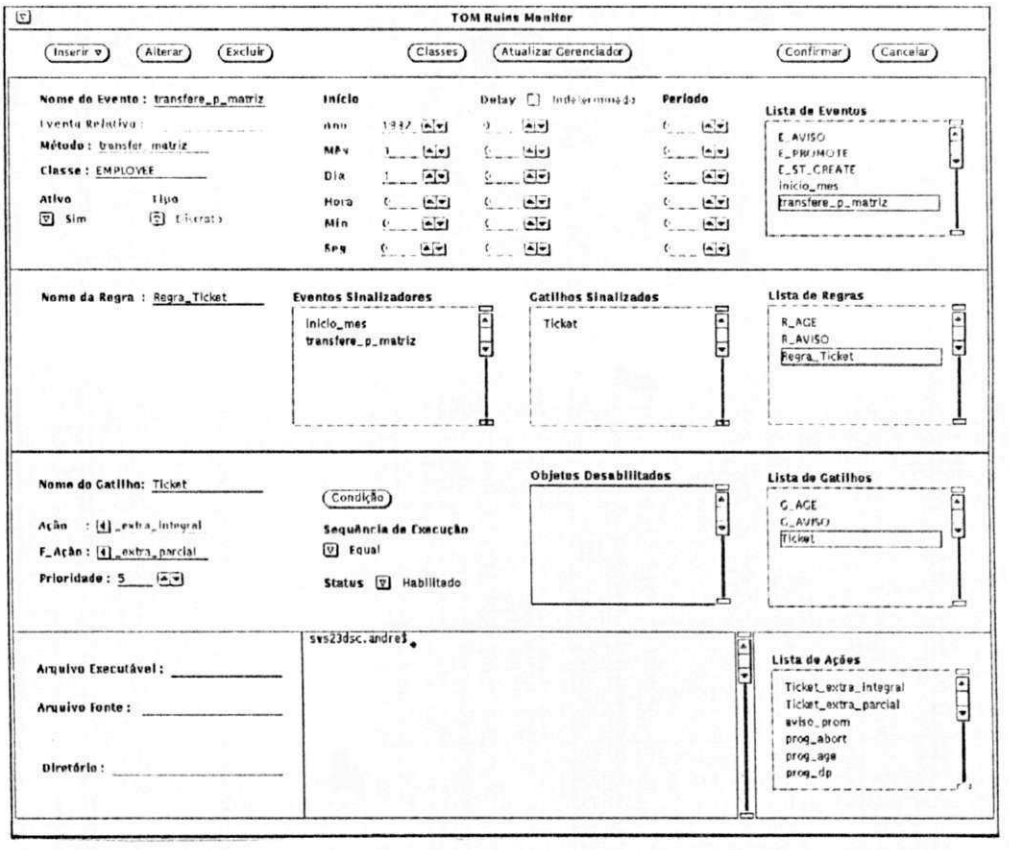


**Figura 5.4.5.9 Janela Base Regra_Ticket**

Para consultar regras eventos e gatilhos basta selecionar o identificador dos mesmos nas respectivas listas. O sistema preencherá todos os campos das respectivas áreas com os valores de cada objeto selecionado. Para alterar um objeto basta selecioná-lo através de um *"click"* de *mouse* sobre seu identificador (na lista respectiva) seguido de um "*click*" sobre o botão *Alterar*. A seguir e só prosseguir com as alterações. Não será permitida a alteração de identificador de objeto dinâmico. A exclusão de um objeto dinâmico será feita através da seleção do mesmo, seguido de um *"click"* sobre o botão *Excluir*. Todas as operações só serão complementadas quando confirmadas através da seleção do botão *Confirma* da *janela base*.

# Capítulo 6

## 6. Conclusão

*No presente capítulo serão apresentadas algumas considerações sobre este trabalho, mais especificamente sobre a modelagem e implementação do TOM Rules. Em seguida são descritas propostas com relação a trabalhos futuros que possam complementar o modelo.*

## 6.1 Considerações sobre o TOM Rules

A maior parte dos sistemas de banco de dados não suportam o armazenamento da dinâmica dos dados, sendo motivação para o desenvolvimento de sistemas de bancos de dados ativos, que possuem esta característica.

Neste trabalho, foi modelado e especificado um módulo que prevê características dinâmicas para o modelo TOM, através da especificação de regras, eventos e gatilhos. Priorizou-se a flexibilidade e robustez na modelagem destes aspectos.

O sistema foi modelado seguindo rigidamente a especificação teórica (excetuando-se a implementação do conhecimento derivado), de modo a provar a viabilidade da abordagem.

Tratar regras, eventos, gatilhos e ações como objetos comuns, trouxe ganhos quanto a flexibilidade, extensibilidade e poder de modelagem dos aspectos dinâmicos, pois separá-los em classes distintas implica numa maior facilidade em especializá-los, isto é, simplifica a construção de objetos dinâmicos mais específicos. Em contrapartida esta abordagem implicou numa redução da garantia de estabilidade do sistema, no que diz respeito à detecção de loops infinitos durante as execuções das ações.

A implementação de regras como restrições de integridade, caracteriza-se por uma limitação funcional, não do modelo, mas do ambiente em que foi desenvolvido o sistema. Desenvolveu-se um eficiente meio de tratamento e detecção de eventos temporais, que merece testes exaustivos com um conjunto de eventos temporais que possuam tempos de ocorrência bem próximos.

Os sistemas ativos estudados utilizam linguagens específicas para construção de aspectos dinâmicos, tornando obrigatório para o usuário o conhecimento da sintaxe destas linguagens. O sistema é provido de uma interface amigável, provendo ao usuário bibliotecas de regras, eventos, gatilhos, ações, e todo um conjunto de facilidades operacionais para manipulação destas bibliotecas. Portanto definir aspectos dinâmicos no TOM Rules consiste de uma tarefa simples onde somente são necessários conhecimentos a respeito do funcionamento de sistemas ativos.

A implementação do modelo proposto foi simplificada graças a utilização da linguagem orientada a objetos Sather, que proveu todos os recursos de modelagem necessários, bem como uma interface simples com a linguagem C e consequentemente com a ferramenta de construção de interfaces GUIDE.

## 6.2 Trabalhos Futuros

Para dar continuidade a este trabalho, vários aspectos podem ser sugeridos com o intuito de tornar o sistema mais completo:

- Extender as condições dos gatilhos de modo que elas não apenas retornem valores booleanos e parâmetros, mas retornem valores complexos resultantes de uma consulta de banco de dados. A interface do sistema foi desenvolvida modularmente, objetivando simplificar a agregação desta característica.
- Implementar um módulo de detecção de loops infinitos nas execuções de ações. Loops infinitos acontecem quando uma regra é composta por um evento de banco de dados que tem uma ação que sinaliza o mesmo evento de banco de dados.
- Implementação do módulo de conhecimento derivado, com uma interface integrada que simplifique sua modelagem, já que neste trabalho nos limitamos a descrevê-lo.
- Migração para um SGBDOO completo, aliviando as restrições do sistema e facilitando a implementação das condições como consultas ao banco de dados

# Bibliobrafia


[AIKE92] Aiken A., Widow J., Hellerstern J. M.:"Behavior of Database Production Rules : Termination, Confluence and Observable Determinism", Proc. ACM SIGMOD, 1992, pp 59-68.

[ANSI86] Burns:"T. Reference Model for DBMS Standartization", ACM SIGMOD, 1986, pp 1958.

[BANC88] Bancilhon F.:"Object Oriented Database System Manifesto", Proc. of the 7th ACM Symposium on the Principle of Database System", Austin, EUA, 1988.

[BANC89] Bancilhon F.,et al:"The Object Oriented Database System Manifesto", Proc of the 1st Conference on Dedutive an Object Oriented Databases", Kyoto, Japão, 1989.

[BAUZ91] Bauzer C., Pfeffer P.:"Transformação de um Banco de Dados Orientado a Objetos para um Banco de Dados Ativo", 6º Simpósio Brasileiro de Banco Dados, Manaus, Amazonas, 1991.

[CHAK89] Chakravarthy S.:"Rule Management and Evaluation of Active DBMS Perspective" SIGMOD Record, Vol.18, Nº3, 1989.

[CHAN82] Chang J., Chang S.:"Database Alerting Techniques for Office Activities Management", IEEE Transactions on Communications, Vol.COM-30, Nº1, 1982.

[DAYA88c] Dayal U., McCarthy D.R.:"The Architecture of an Active Database Management System", Proc. ACM SIGMOD, Portland, EUA, 1989, pp 215-224.

[DAYA88b] Dayal U. et al:"The HIPAC Project : Combining Active Databases and Timming Constraints", SIGMOD Record, Vol.17, Nº1, 1988.

[DAYA88a] Dayal U., Buchanan A.P., McCarthy D.R.:"Rules are Objects Too : A Knowledge Model for Active Object Oriented Database Systems", Proc. of 2nd International Workshop on Object Oriented Database Systems, LNCS 334, Spring Verlag, 1988, pp 129-143.

[DAVI91] David M.B.:"Descrição Formal da Estrutura do Modelo Orientado a Objetos Temporal TOM (Temporal Object Model)", Tese de Mestrado, DSC - UFPb, Campina Grande, 1992.

[DEUX89] Deux O., et al:"The Story of O2", Altair Technical Report, 1989.

[DIAZ91] Diaz O., Paton N., Gray P.:"Rule Management in Object Oriented Databases: A Uniform Approach", Proc. 17th VLDB, Barcelona, 1991, pp 317-326.

[DITT86] Dittrich K., Kotz A., Muller J.:"Supporting Semantic Rules by Generalized Event/Trigger Mechanism", Proc. 1st. EDBT, 1988, pp 76-91.

[GEHA91] Gehani N., Jagadish H.V.:"Ode as an Active Database : Constraints and Triggers", Proc, 17th. VLDB, Barcelona, 1991, pp 327-336.

[KIM 89] Kim W., Lochovsky F.H.:"Object Oriented Concepts, Databases and Applications", Addison-Wesley, 1989.

[KHOS89] Khoshafian S., Parsaye K., Wong H.:"Intelligent Databases - Object Oriented, Dedutive and Hypermedia Technologies", John Willey Sons, Inc., 1989.

[LANC92] Lancastre M.:"BR+: Um Modelo de Dados para um Ambiente Multimídia", Tese de Mestrado, Departamento de Informática - UFPE, Recife, 1992.

[LECL89] Lecluse C., Richard P., Vélez F.:"The O2 Data Model", Altair Technical Report, 1989.

[LING87] Lingart J. Y., Nobecourt P., Rolland C.:"Behavior Management in Database Applications", Proc. VLDB Conference, Brighton, Inglaterra, 1987, pp 185-196.

[MEIR92] Meira S.R.L., Fernandes J.H.C:"GOD - Um Gerenciador de Objetos Distribuídos para Ambientes de Desenvolvimento de Software", VI Simpósio Brasileiro de Engenharia de Software, Gramado, 1992, pp 273-291.

[OMHU91] Omhundro S.M.:"The Sather Language", Sather Language Manpage, Berkeley, EUA, 1991.

[PROT86] Schiel U. et al.:"Ambiente Aberto para Desenvolvimento de Banco de Dados Ativos Distribuídos", Relatório PROTEM - Universidade Federal da Paraíba, Campina Grande, 1992.

[ROLL89] Rolland C., Cauvet C.:"A Design Methodology for Object Oriented Database Systemas" in Proc. Conference on Management of Data (COMAD), 1989, pp 39-65.

[SCHI83] Schiel U.: "An Abstract Introdution to Temporal Hierarchic Data Model (THM)", Proc. 9th. VLDB, Florence, 1983.

[SCHI91] Schiel U.: "An Open Environment for Objects with Time and Versioning", proc EastEurOOPe, Bratislava, 1991.

[STON86] Stonebraker M., Rowe L. A.:"The Design of Postgres", Proc. ACM SIGMOD Conference of Management of Data, Washington, EUA, 1986.

[STON87] Stonebraker M., Rowe L.A:"The Postgres Data Model" in Proc, VLDB Brighton, Inglaterra, 1987.

[STON88] Stonebraker M., Hanson E. N., Potamianos S.:"The Postgres Rule Management", IEEE Software Engeneering, Vol.14, Nº7, 1988, pp 897-907.

[STON90] Stonebraker M., et al.:"On Rules Procedures, Caching and Views in Data Base Systems", Proc. ACM SIGMOD Conference on Management of Data, Atlantic City, EUA, 1990.

[STON91] Stonebraker M., Kemnitz G.:"The Postgres Next Generation Database Management System", Communications of the ACM Vol.34, Nº10, 1991, pp 78-92.

[TADA91] Takahashi T., Liesenberg H.K.E.:"Programação Orientada a Objetos", VII Escola de Computação, IME-USP, São Paulo, 1991.

[URBA90] Urban S. D., Delcambre L. M. L.:"Constraints Analysis : A Design Process for Specifying Operations in Objects", IEEE Transactions on Knowledge and Data Engeneering, Vol.2, Nº4, 1990, pp 391-400.