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

TF-ORM: O MODELO E UMA ESPECIFICAÇÃO

por
Nina Edelweiss
Nicole Silva de Freitas
Érico Martelet Marcant

RP-258 Março/96
Relatório de Pesquisa

UFRGS - II - CPGCC
Caixa Postal 15064 - CEP 91501-970
Porto Alegre - RS- Brasil
Telefone: (051) 336-8399 e 339-1355
Fax: (051) 336-5576
Email: PGCC@INF.UFRGS.BR

RESUMO

Verifica-se, atualmente, a necessidade de especificação e implementação de sistemas com a utilização de aspectos temporais, principalmente em sistemas de informação de escritórios e aplicações industriais onde as características temporais são extremamente relevantes.

Neste trabalho, é apresentada uma das formas utilizadas para a especificação de sistemas que é a definição de um esquema conceitual, utilizando um modelo de dados que represente as características dinâmicas e a interação temporal entre diferentes processos dentro da aplicação. Optou-se por utilizar um modelo de dados temporal orientado a objetos o qual permite a representação de aspectos temporais tanto para a definição de dados como para representar a evolução dos valores assumidos pelos objetos durante sua existência.

O modelo utilizado denomina-se TF-ORM (*Temporal Functionality in Objects with Roles Model*) e é um modelo orientado a objetos, temporal, que utiliza o conceito de papéis para representar os diferentes comportamentos de um objeto.

São apresentados a sintaxe do modelo TF-ORM e um estudo de caso com o objetivo de exemplificar a modelagem TF-ORM em sistemas de informação.

PALAVRAS-CHAVE: Modelagem Temporal, Modelo de Dados Orientado a Objetos.

ABSTRACT

Nowadays, the necessity of specification and implementation of systems using temporal aspects is noticed, specially in office information systems and industrial applications where temporal characteristics are extremely relevant.

In this work, a method of system specification is presented - the definition of a conceptual scheme, using a data model able to represent the dynamic characteristics and the temporal interaction between differents processes in the application. A Temporal Object-Oriented data model was chosen. This model allows the representation of temporal aspects for data definition and of the evolution of the object values during its existence.

The used model is called TF-ORM (*Temporal Functionality in Objects with Roles Model*) and is an object-oriented temporal data model that uses the concept of roles to represent the different object behaviors.

The TF-ORM syntax is presented and a complete case study is developed to exemplify the TF-ORM modeling of information systems.

KEYWORDS: Temporal Modeling, Object-Oriented Data Model.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

1. INTRODUÇÃO

A implementação de um sistema de informação apresenta diferentes fases a serem executadas, sendo a especificação de requisitos do sistema a ser implementado uma das mais importantes. Essa especificação, ou modelagem da aplicação, se baseia na coleta de requisitos a partir do mundo real, procurando a identificação e descrição de todas as características que o sistema apresentará. Deve, contudo, ser totalmente independente da implementação, servindo de base para a realização da mesma.

Na modelagem de sistemas de informação, a modelagem de aspectos temporais é um importante tópico, já que através destes aspectos são representadas as características dinâmicas das aplicações e a interação temporal entre diferentes processos. A possibilidade de armazenar, manipular e recuperar dados temporais deve ser considerada no momento da escolha de um método de modelagem. Por esse motivo, considerando modelos de dados orientados a objetos, foi escolhido o modelo de dados **TF-ORM** (*Temporal Functionality in Objects with Roles Model*) como objeto de estudo.

O modelo TF-ORM apresenta tanto aspectos temporais como os próprios do paradigma de orientação a objetos, tais como: (i) a hierarquia de classes; (ii) a recuperação de atributos representados por classes; e (iii) a existência de diversas instâncias de uma classe identificada. Apresenta, ainda, o conceito de *papéis* para representar separadamente diferentes comportamentos dos objetos de uma classe, facilitando o processo de análise da aplicação e a representação da evolução dos objetos através do tempo.

O objetivo principal deste trabalho é o estudo do modelo de dados TF-ORM através da modelagem de um estudo de caso. Assim, possibilita-se um maior entendimento sobre a especificação de sistemas de informação, ocasionando, também, um maior entendimento sobre banco de dados orientados a objetos.

Inicialmente, no capítulo 2, são apresentados os conceitos básicos do modelo TF-ORM. A sintaxe completa da linguagem de definição de dados do modelo é apresentada no Anexo 1.

No capítulo 3, descreve-se o problema selecionado para o estudo de caso, bem como aspectos principais do funcionamento da solução proposta.

A modelagem utilizada para exemplificar o modelo de dados TF-ORM em especificações de sistemas de informação de escritórios é apresentada no capítulo 4.

2. MODELO TF-ORM

2.1. O Conceito de Papéis

Em **modelos de dados orientados a objetos convencionais** um objeto é criado como uma instância de uma classe, apresentando as propriedades e comportamento desta classe. O objeto apresenta um identificador único que o distingüe dos demais objetos. Uma vez criado, o objeto pertence a esta classe durante todo o seu tempo de vida, mesmo se, com o passar do tempo, suas características comportamentais evoluam de modo a ser identificado com as características de outra classe.

Como exemplo deste tipo de evolução, considere a classe *pessoa* com duas subclasses - *estudante* e *funcionário* (Figura 2.1). Um objeto pode ser instanciado em somente uma das subclasses, o que significa que ele pode ser estudante ou funcionário. Não é possível representar o fato de um determinado estudante deixar de ser estudante e passar a ser funcionário, o que representaria uma migração do objeto entre subclasses.

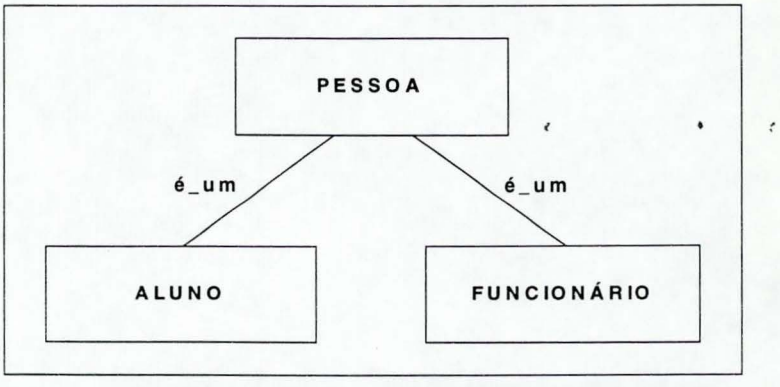

Figura 2.1: Modelo de dados OO tradicional

Alguns modelos permitem algum tipo de migração de objetos entre classes, como no modelo KNOs [CAS 88, TSI 87] e PROBE [MAN 86], sendo esta migração geralmente restringida entre classe e subclasse.

Outra característica de modelos de dados orientados a objetos tradicionais é não permitir que um objeto pertença simultaneamente a mais de uma classe. No exemplo anterior este fato seria necessário para representar o caso de um estudante que passa a ser funcionário, continuando como estudante. Também não é possível um objeto ser representado por múltiplas instâncias de uma mesma classe, como no caso em que se quer representar uma pessoa que apresente mais de um emprego, com características diferentes (empregador, data de admissão, salário, próprios de cada um dos empregos).

A utilização do **conceito de papéis** resolve estas limitações dos modelos orientados a objetos tradicionais, permitindo a representação da evolução do comportamento de um objeto. Através deste conceito, uma classe pode apresentar diversos papéis, os quais podem ser instanciados dinamicamente. Um objeto continua sendo instância de uma só classe, mas pode desempenhar diferentes papéis durante sua existência. Os papéis desempenhados por um objeto são representados por instâncias destes papéis, as quais podem ser dinamicamente criadas e destruídas durante a existência do objeto. Um objeto pode apresentar simultaneamente instâncias de mais de um papel, assim como pode apresentar mais de uma instância de um mesmo papel no mesmo instante de tempo.

No exemplo acima, em vez de utilizar subclasses para representar os comportamentos de *estudante* e *funcionário* seriam utilizados papéis contidos na definição da classe *pessoa*. A abstração de generalização/classificação continua existindo, estendendo-se o conceito de herança também para os papéis (Figura 2.2). Um objeto pessoa que fosse somente estudante apresentaria uma instância do papel estudante. A evolução do objeto trocando de papel pode ser representada através de criação de instância de outro papel e de destruição da instância existente. O fato de um estudante ser também funcionário seria representado por uma instância de estudante e uma instância de funcionário, as duas existindo simultaneamente. E para a pessoa que apresente mais de um emprego, cada um destes seria representado por uma instância independente deste papel.

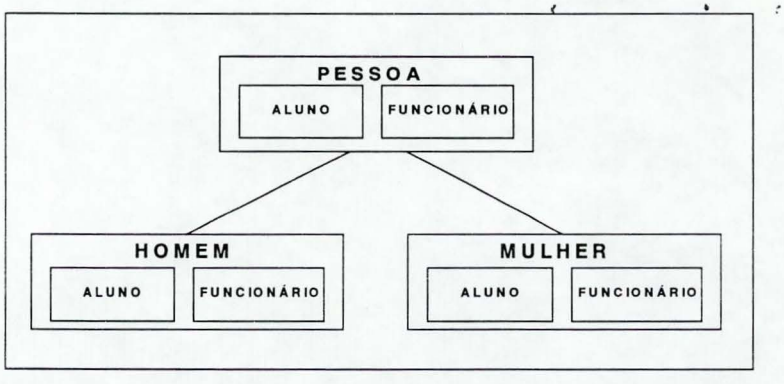

Figura 2.2: Modelo orientado a objetos com papéis

2.2. Origem do Modelo TF-ORM

TF-ORM (*Temporal Functionality in Objects with Roles Model*)[EDE 93a,b, 94] é um modelo de dados temporal orientado a objetos, que utiliza o conceito de papéis para representar diferentes comportamentos de um mesmo objeto.

O conceito de papéis associado à orientação a objetos foi introduzido através do modelo ORM (*Objects with Roles Model*) [PER 90]. A partir do modelo ORM foi feita uma primeira extensão, o modelo F-ORM (*Functionality in Objects with Roles Model*) [DEA 91], na qual se procurou representar as funcionalidades de sistemas de informação de escritórios. Novas extensões foram realizadas para permitir a representação de aspectos temporais (propriedades temporais e evolução temporal de

objetos) e de atividades não estruturadas (decisões humanas), dando origem ao modelo TF-ORM.

Nos itens seguintes serão apresentadas as características do modelo TF-ORM e sua linguagem de especificação.

2.3. Informações Temporais

O modelo TF-ORM foi criado com o objetivo de armazenar não somente os valores atuais dos dados, mas toda a sua história - valores passados, atuais e previsões de valores futuros. Todos os estados assumidos por uma instância são permanentemente armazenados no banco de dados, com rótulos temporais que indicam quando foram efetuadas as definições. Com este objetivo, informações temporais foram associadas às informações, em dois níveis: (1) às instâncias, através de propriedades especiais (*object_instance* e *role_instance*), representando a evolução das instâncias como um todo; e (2) às propriedades que possam variar com a passagem de tempo, identificando quando os valores foram definidos e quando se tornaram válidos.

Dois diferentes conceitos de tempo podem ser identificados em uma aplicação [JEN 94, SNO 85]:

- *tempo de transação* - o tempo em que uma informação foi introduzida no banco de dados; e
- *tempo de validade* - tempo em que a informação modela a realidade, ou seja, o tempo em que a informação tem validade no mundo real.

No modelo TF-ORM ambos os tempos acima são utilizados, utilizados como rótulos associados às propriedades que descrevem as instâncias e às propriedades que variem com o tempo. O tempo de transação é implicitamente definido pelo SGBD. O tempo de validade deve ser fornecido pelo usuário, representando o instante de início da validade da informação. Este valor permanecerá válido até que outra informação seja definida e se torne válida.

O tempo é modelado como variando de forma discreta, estando linearmente ordenado. Quatro formas diferentes podem ser utilizadas como elemento temporal primitivo para a representação explícita de tempo [ADI 87]: o ponto no tempo (instante temporal), o intervalo temporal, a duração e tempo periódico. No modelo TF-ORM foi selecionado o *ponto no tempo* como elemento temporal primitivo, sendo possível representar as outras formas através dele. Para definir completamente um ponto no tempo é necessário definir a data e horário correspondentes ao instante temporal.

Em virtude das aplicações que se pretende modelar através deste modelo (sistemas de informação de escritórios, sistemas de informações médicas, empresas), a menor granularidade temporal utilizada no modelo é o *minuto*, medida mínima para medir atividades humanas. O minuto foi, portanto, escolhido como *chronon* [JEN 94] do modelo.

2.4. Definição de Classes e de Papéis

Nos próximos itens será apresentada a linguagem de definição do modelo TF-ORM. A BNF completa desta linguagem encontra-se no Anexo 1.

Uma **classe** é definida através de um nome c_n e de um conjunto de papéis R_i, cada um representando um comportamento diferente dos objetos desta classe:

$$classe = (c_n, R_0, R_1, ..., R_n)$$

Ao definir uma classe, deve ser também definido o seu tipo. São três os **tipos de classes**: classe de agente (*agent class*), de recurso (*resource class*) ou de processo (*process class*). Classes de agentes representam pessoas atuando no sistema que está sendo modelado, e cada papel representa um comportamento que esta pessoa pode apresentar na aplicação considerada. As estruturas dos recursos (dados, documentos) são representadas através de classes de recursos, com os papéis que os recursos podem desempenhar durante a sua existência. As classes de processos integram estes agentes e recursos, descrevendo a organização do trabalho desenvolvida na aplicação, e a cooperação entre os agentes. Os papéis nas classes de processo representam diferentes tarefas a serem executadas no processo.

Cada **papel** é constituído de um nome Rn_i, de um conjunto de propriedades P_i deste papel (descrições abstratas dos tipos de dados implementados como variáveis de instâncias), de um conjunto de estados S_i que este objeto pode assumir quando desempenhando este papel, de um conjunto de mensagens M_i que o objeto neste papel pode receber e enviar, e de um conjunto de regras Ru_i (regras de transição entre estados e regras de integridade):

$$R_i = < Rn_i, P_i, S_i, M_i, Ru_i >$$

Além das mensagens acima mencionadas, as classes de agentes podem também apresentar um conjunto de decisões D_i, através das quais é modelada a parte não estruturada das aplicações, de uma maneira formal:

$$R_i = < Rn_i, P_i, S_i, D_i, M_i, Ru_i >$$

Todas as instâncias de papéis evoluem independentemente, podendo haver interação entre elas através do envio de mensagens.

Dois tipos de instâncias são, portanto, utilizados no modelo TF-ORM: instâncias de classes (objetos) e instâncias de papéis. Uma instância de uma classe (um objeto) pode apresentar diversas instâncias de papéis. Os dois tipos de instâncias apresentam identificadores únicos - identificadores de objetos e identificadores de instâncias de papéis.

Todo objeto apresenta um papel especial R_0, denominado papel básico (*base role*). Neste papel são descritas as características inciais de toda instância desta classe e as suas propriedades globais. Este papel apresenta somente propriedades (herdadas por todas as instâncias dos demais papéis) e regras (que controlam as instâncias dos outros papéis). Seus estados são pré-definidos (*active* e *suspended*), assim como as mensagens que podem ser utilizadas em suas regras (mensagens de criação, suspensão, retomada e término de instância de uma classe e de um papel).

2.5. Propriedades e Mensagens Pré-Definidas

TF-ORM apresenta uma classe denominada *OBJECT*, que atua como superclasse para todas as classes definidas. O papel básico de *OBJECT* contém três **propriedades pré-definidas**, herdadas por todos os papéis-base das classes definidas:

- *oId* - armazena o identificador de instância de objeto;
- *object_intance* - armazena todas as informações relativas à evolução temporal do objeto (instante de sua criação, tempos em que o objeto esteve suspenso e em que retomou sua atividade); e
- *object_end* - armazena a informação relativa ao instante da destruição do objeto.

A existência de uma instância de uma classe inicia no momento de sua criação, sendo este instante de tempo armazenado na propriedade *object_instance*, com valor *nonull* indicando que a instância está ativa. Se esta instância for temporariamente suspensa, a informação temporal relativa ao início da suspensão é armazenada nesta mesma propriedade, com valor *null* indicando a sua suspensão. A retomada de sua atividade é também registrada nesta propriedade, voltando seu valor a ser *nonull*.

A superclasse *OBJECT* apresenta um único papel, denominado *ROLE*, no qual estão especificadas três propriedades pré-definidas, herdadas por todos os outros papéis das classes definidas (com excessão do papel básico):

- *rId* - armazena o identificador da instância do papel;
- *role_instance* - armazena todas as informações relativas à evolução temporal da instância do papel (instante de criação, tempos em que esteve suspenso e em que retomou a sua atividade); e
- *role_end* - armazena a informação relativa ao instante da destruição dessa instância do papel.

A existência de uma instância de um papel depende da existência da instância da classe na qual está contida. Portanto, os valores contidos em *object_instance* e *object_end* são restrições temporais impostas àqueles contidos em *role_instance* e *role_end*. O término da existência de uma instância de classe faz com que todas as instâncias de papéis nela contidos tenham também suas existências terminadas.

Deste modo, toda instância de uma classe apresenta a propriedade *oId* na qual é armazenado o identificador interno desta instância. E toda instância de um papel apresenta a propriedade *rId* onde é armazenado o identificador interno desta instância. Os valores contidos nestas propriedades não são acessíveis, mas podem ser referenciados nas condições associadas às regras de transição de estados, nas regras de integridade, nos métodos que implementam as mensagens e na linguagem de consulta do modelo.

O papel básico de *OBJECT* apresenta ainda um conjunto de **mensagens pré-definidas**, as quais são herdadas por todas as classes definidas, e que podem ser utilizadas tanto pelos papéis-base como pelos demais papéis definidos. São elas:

- *create_object (C, Oid)* - cria uma instância da classe *C*, cujo identificador é devolvido pelo sistema através do argumento *OId*;
- *suspend_object (Oid)* - suspende temporariamente a atividade da instância de classe cujo identificador é fornecido. Esta instância não pode enviar nem receber mensagens, com excessão das mensagens que indicam a retomada de suas atividades ou a sua destruição;
- *resume_object (Oid)* - permite à instância de classe identificada por *Oid* a retomada de suas atividades;
- *kill (Oid)* - termina a existência de uma instância de classe;
- *allow_role (Oid, R)* - permite a criação de instâncias de um determinado papel da classe identificada por *Oid*. Não é necessária a identificação da classe quando a mensagem for utilizada na mesma classe em que está sendo criada a instância do papel;
- *add_role (Oid, R, Rid)* - cria uma instância do papel *R* da instância de classe identificada por *Oid*, sendo o identificador desta instância de papel devolvido através do argumento *Rid*;
- *suspend_role (Rid)* - suspende temporariamente as atividades da instância de papel identificada por *Rid*;
- *resume_role (Rid)* - retoma as atividades da instância de papel *Rid*;
- *terminate_role (Rid)* - termina a existência da instância de papel *Rid*;
- *forbid_op (Rid, Direção, Mensagem)* - impede que uma determinada mensagem seja temporariamente enviada ou recebida pela instância de papel *Rid*; e
- *allow_op (Rid, Direção, Mensagem)* - permite novamente o envio ou recebimento de uma mensagem que tinha sido desabilitada.

2.6. Definição de Propriedades

2.6.1. Tipos de Propriedades

O modelo TF-ORM apresenta dois tipos diferentes de propriedades, conforme a possibilidade de apresentarem variação com o tempo. Uma análise das possíveis propriedades que um objeto pode apresentar mostra que existem algumas que mantém o mesmo valor durante toda a sua existência (por exemplo, a data de nascimento de uma pessoa), enquanto que outras podem ter valores que variam com o passar do tempo (como o salário de um funcionário). As propriedades cujos valores não variam são denominadas no modelo TF-ORM de **propriedades estáticas** (*static properties*), e as que podem variar, de **propriedades dinâmicas** (*dynamic properties*). Todos os valores definidos para as propriedades dinâmicas devem ser armazenados no banco de dados, podendo ser recuperados em eventuais consultas.

As propriedades estáticas e dinâmicas são tratadas de formas diferentes no modelo TF-ORM. As propriedades estáticas não apresentam rótulos temporais. No instante de criação da instância, todas as suas propriedades estáticas são definidas, com valor inicial *null*. Este valor é alterado apenas uma vez, quando for feita a definição do valor da propriedade, valor este que é considerado válido para toda a existência da instância.

Uma propriedade dinâmica consiste de um conjunto de triplas, apresentando cada uma a seguinte forma (exemplo do salário de um funcionário):

salário: REAL X INSTANT X INSTANT

O valor correspondente à propriedade (neste exemplo REAL) é associado a dois rótulos temporais (INSTANT X INSTANT) correspondendo aos tempos de transação e de validade.

2.6.2. Domínios de Propriedades

Para toda propriedade (estática ou dinâmica) deve ser definido um domínio sobre o qual seus valores podem ser definidos. Os domínios podem ser simples ou complexos. Podem ser utilizados como domínios um objeto, um conjunto de objetos e uma lista de objetos. TF-ORM apresenta um conjunto de domínios pré-definidos, denominados *tipos de dados*, apresentados na tabela 2.1.

Tabela 2.1 : Domínios Pré-definidos para Definição de Propriedades

integer	**instant**	**week**
real	**date**	**semester**
boolean	**time**	**century**
string	**year**	**weekday**
text	**month**	**interval** (Tipo Intervalo , Tipo Limites)
place	**day**	**span** (Tipo Temporal)
title	**hour**	**after** (Tipo Temporal)
image	**minute**	**before** (Tipo Temporal)
		within (Tipo Limites)

O domínio *instant* é composto da concatenação de data (dia, mês e ano) e hora (hora e minuto). Este tipo de dado temporal é utilizado nas propriedades dinâmicas, para rotular os tempos de transação e de validades.

Os domínios temporais intervalo (*interval*), duração (*span*), depois de (*after*), antes de (*before*) e dentro (*within*) necessitam de informações adicionais em sua definição. Ao associar um intervalo como domínio de uma propriedade é necessário definir qual o tipo deste e qual o tipo dos seus limites. Os tipos de intervalo são os seguintes:

- *closed* - fechado, ou seja, quando ambos os limites pertencem ao intervalo;
- *open* - aberto, ou seja, quando nenhum dos limites pertence ao intervalo;
- *open_down* - quando somente o limite inferior não pertence ao intervalo;
- *open_up* - quando somente o limite superior não pertence ao intervalo;

- *floating_down* - quando o limite inferior é o momento atual (*now*); e
- *floating_up* - quando o limite superior é o momento atual.

Como tipos de limites de intervalos podem ser utilizados *instant*, *date*, *time*, *year*, *month*, *day*, *hour* e *minute*.

Para os domínios temporais *span*, *after* e *before* deve ser informado qual o tipo de domínio temporal em que vai ser feita a medida da duração ou a comparação temporal. Podem ser utilizados para isto os seguintes tipos temporais: *year*, *month*, *day*, *hour*, *minute*, *week*, *semester* e *century*.

Os domínios temporais *after*, *before* e *within* são utilizados para a representação de informações temporais não bem definidas - o ponto no tempo não é perfeitamente determinado, somente o intervalo dentro do qual ele deve aparecer. Representam um ponto no tempo, com restrição para o intervalo de tempo em que pode ser definido. *After* e *before* modelam pontos no tempo que podem ser definidos antes ou depois de uma determinada data - dentro de um intervalo fechado de um lado. *Within* é utilizado para um ponto no tempo que pode ser definido dentro de um determinado intervalo.

Entre os domínios pré-definidos acima, diversos são para a representação de informações temporais, com diferentes granularidades temporais. A utilização de diferentes granularidades temporais pode trazer dificuladades na manipulação e operação destas informações [CLI 88, WIE 91]. Para resolver estes problemas foi definido, em TF-ORM, um conjunto de funções (predicados) para serem utilizados em condições lógicas da modelagem (restrições e regras de transição) e na linguagem de consulta. Estas funções convertem tipos de diferentes granularidades temporais, permitindo operações entre eles. Três tipos de funções estão disponívies: (i) funções que trocam a granularidade temporal; (ii) funções que retornam uma informação temporal extraída de um valor temporal, tal como *month* (<instant | date>) que extrai o mês de um instante ou de uma data; e (iii) funções lógicas que comparam a posição temporal relativa de duas informações temporais de mesma granularidade, como *before* (<instant>:<instant>) que retorna verdadeiro quando o primeiro instante preceder o segundo no tempo. O conjunto completo de funções definidas pode ser encontrado na BNF do Anexo 1.

2.7. Definição de Estados

Os estados de um papel representam estados abstratos que uma instância deste papel pode assumir durante sua existência. Por exemplo, um funcionário pode estar nos estados "trabalhando, de_férias, de_licença, aposentado".

A definição dos estados de um papel é feita listando os nomes dos estados. Estes nomes serão depois utilizados nas regras de transição de estados, onde é feita a modelagem da evolução dos mesmos.

Os nomes dos estados devem ser únicos dentro de um papel - papéis diferentes podem apresentar nomes de estados iguais.

2.8. Definição de Mensagens

Todas as mensagens que uma instância de um papel pode enviar ou receber devem ser listadas, com seus parâmetros e origem/destino.

Na lista de parâmetros deve ser definido um domínio para cada parâmetro, podendo ser utilizado qualquer dos domínios definidos para as propriedades.

Mensagens podem ser enviadas ou recebidas de classes ou de papéis. A identificação de receptor/emissor é feita através do nome da classe (opcional, caso o contexto deixe isto claro) e do papel.

Uma mensagem recebida de uma classe seria enviada pelo papel básico desta classe. Uma mesma mensagem pode ser recebida de mais de um emissor, e pode ser enviada simultaneamente a mais de um receptor. A definição de que uma mensagem pode ser recebida de mais de um papel significa que somente um destes papéis necessita enviar a emnsagem para que sua regra de transição seja ativada. Já no envio o conceito é outro - ao ser ativada por uma regra de transição, cópias desta mensagem são enviadas a cada um dos papéis definidos como destino.

Uma mensagem pode ser enviada por uma instância de um papel a si mesmo (*to itself*).

Quando o emissor/receptor da mensagem não estiver definido na modelagem, a mensagem deve ser declarada como tendo o mundo externo (*External_World*) como interface. Deste modo é possível modelar aplicações em vários níveis de detalhamento, deixando o que deve ser detalhado mais adiante como sendo representado pelo mundo externo.

2.9. Definição de Decisões

As classes de agentes podem apresentar um conjunto de decisões. Uma decisão pode apresentar parâmetros, devendo para cada um ser definido um domínio.

2.10. Definição de Regras

Dois tipos de regras podem ser definidas em TF-ORM: regras de transição de estados e regras de integridade. Todas as regras recebem um nome, nome este que é utilizado somente para facilitar a modelagem - através deste nome pode-se identificar qual o funcionamento da regra.

2.10.1. Regras de Transição de Estados

As regras de transição de estados definem quais são as mudanças de estados que podem ocorrer em instâncias de um papel. A forma mais geral de uma regra de transição de estados é a seguinte:

$$r_n: state(s_1), msg(mi) \Rightarrow msg(mo), state(s_2); (\ <condição\ de\ transição>\ )$$

Esta regra tem o seguinte significado: quando a instância recebe a mensagem *mi*, se esta instância estiver no estado s_1 e se a condição de transição for satisfeita, então esta instância envia a mensagem *mo* e muda de estado, assumindo o estado s_2.

Nenhum dos elementos acima é obrigatório na definição de uma regra de transição de estados. As seguintes combinações podem ocorrer:

- o estado inicial s_1 não é definido - a regra será ativada sempre que chegar a mensagem *mi*, se a condição for verdadeira, independentemento do estado que a instância estiver desempenhando;
- a mensagem de chegada *mi* não é definida - a regra será ativada sempre que a instância estiver no estado s_1, devendo a condição ser verdadeira;
- a mensagem de saída *mo* não é definida - ocorre uma transição de estado sem envio de mensagem;
- o estado final s_2 não é definido - a regra, quando ativada, envia alguma mensagem sem que ocorra uma transição de estado;
- a condição não é definida - a ativação da regra fica restrita à chegada da mensagem estando a instância no estado inicial.

Duas formas alternativas desta forma podem ser utilizadas, relativas à possibilidade de utilização de múltiplas mensagens de entrada e de saída. Assim, a regra a seguir:

$$r_n: state(s_1),\ msg(mi) \Rightarrow msg(mo_1),\ msg(mo_2),\ \ldots,\ msg(mo_n),\ state(s_2);$$

$$(\ \langle condição\ de\ transição \rangle\ )$$

define que, ao ocorrer uma transição, não somente uma mensagem, mas um conjunto de mensagens, é enviado. E a regra:

$$r_n: state(s_1),\ \{msg(mi_1),\ msg(mi_2),\ \ldots,\ msg(mi_m)\} \Rightarrow msg(mo),\ state(s_2);$$

$$(\ \langle condição\ de\ transição \rangle\ )$$

define que a transição vai ocorrer somente depois que chegarem todas as mensagens $mi_1, mi_2, \ldots, mi_m$. A chegada destas mensagens pode ocorrer em qualquer ordem, em tempos diferentes.

Uma regra de transição pode ser acionada por uma tomada de decisão:

$$r_n: state(s_1),\ decision(d_1) \Rightarrow msg(mo),\ state(s_2);\ (\ \langle condição\ de\ transição \rangle\ )$$

A forma de expressar a condição de transição será explicada na seção 2.9.3.

2.10.2. Regras de Integridade

As regras de integridade devem ser sempre satisfeitas por todas as instâncias do papel. Apresentam a seguinte forma:

$$constraint\ (\ \langle pré\text{-}condição \rangle \Rightarrow \langle pós\text{-}condição \rangle\ )$$

Sempre que a pré-condição de uma regra de integridade for satisfeita, sua pós-condição também deve ser. Deste modo, estados inconsistentes do banco de dados não são permitidos. As regras de integridade de um papel devem ser verificadas a cada vez que uma regra de transição de estados for ativada - se for constatada a quebra da integridade do banco de dados, esta transição deverá ser desfeita, restaurando os valores anteriores das propriedades modificadas pelo(s) método(s) que implementa(m)

a(s) mensagem(s) recebida(s). Além disso, deve ser evitado que a(s) mensagem(s) de saída seja(m) enviada(s).

2.10.3. Linguagem de Definição das Condições

As condições definidas nas regras de transição de estados e nas regras de integridade devem ser expressas através de sentenças de uma linguagem de lógica temporal de primeira ordem. Nestas condições podem ser avaliados valores de propriedades armazenados no banco de dados (presentes, passados e futuros) e estados de papéis. Os identificadores de instâncias de classes e de instâncias de papéis podem também ser referenciados, pois estão armazenados nas propriedades pré-definidas *oId* e *rId*.

Os *símbolos* que podem ser utilizados em sentenças desta linguagem são variáveis, constantes, predicados e funções.

O *alfabeto A* da linguagem apresenta, além dos símbolos lógicos usuais em qualquer linguagem de primeira ordem (operadores aritméticos, operadores lógicos, quantificadores aplicados a variáveis individuais), mais um conjunto de operadores temporais (Tabela 2.2) e um conjunto especial de predicados não lógicos (Tabela 2.3) e de funções (Tabela 2.4).

Tabela 2.2 - Operadores Temporais

sometime past
immediately past
always past
sometime future
immediately future
always future
since
until
before
after

Tabela 2.3 - Símbolos Predicados

Símbolo Predicado	Sorte
has_class_instance	<nome da classe> <instância>
has_role_instance	<instância> <nome do papel> <instância>
active_class	<instância>
active_role	<instância>
active_class_at	<instância> <instante temporal>
active_role_at	<instância> <instante temporal>
is_valid	<instância> <nome da propriedade>
is_valid_at	<instância> <nome propriedade> <instante temporal>
out_role	<instância> <nome do papel>
role	<instância> <instância do papel>
begin	<nome da propriedade>
contains	<nome de propriedade> <nome de propriedade>

Tabela 2.4 - Funções Temporais

Funções Temporais	Sortes
value	<instância> <nome de propriedade>
past_value	<instância> <nome propriedade> <instante temporal>
valid_time	<instância> <nome de propriedade>
transaction_time	<instância> <nome de propriedade>
class_creation_time	<instância>
role_creation_time	<instância>
class_end_time	<instância>
role_end_time	<instância>
state	<instância>
state_at	<instância> <instante temporal>
year	<instante temporal>
month	<instante temporal>
day	<instante temporal>
hour	<instante temporal>
minute	<instante temporal>
weekday	<instante temporal>
lower_bound	<intervalo temporal>
upper_bound	<intervalo temporal>
duration	<duração temporal>

A seguir estão detalhados os significados de alguns dos predicados e funções da linguagem. São utilizadas palavras iniciando por maiúsculas para indicar variáveis:

- *has_class_instance(Classe, OId)* - retorna verdadeiro quando existe pelo menos uma instância da classe *Classe*, sendo o identificador desta instância devolvido através de *OId*. Este predicado retorna somente uma instância da classe;
- *has_role_instance(OId, Papel, RId)* - retorna verdadeiro quando existe pelo menos uma instância do papel *Papel* da instância da classe identificada por *Oid*. Retorna o identificador da instância do papel em *Rid*. Somente uma instância do papel é retornada;
- *value(RId, Propriedade)* - retorna o valor atualmente válido da propriedade *Propriedade* da instância identificada por *Rid*. O *Rid* não precisa ser fornecido quando a função retornar o valor de uma propriedade do mesmo papel em que está escrita a condição;
- *past_value(RId, Propriedade, Tempo)* - retorna o valor da propriedade *Propriedade* da instância *Rid* no instante de tempo *Tempo*;
- *valid_time(RId, Propriedade)* - retorna o tempo em que iniciou a validade do valor atual da propriedade *Propriedade* da instância *Rid*;
- *class_creation_time(OId)* - retorna o instante de tempo correspondente à criação da instância da classe identificada por *Oid*;
- *role_creation_time(RId)* - retorna o instante de tempo correspondente à criação da instância de papel identificada por *Rid*;
- *state(Id)* - retorna o atual estado da instância de classe ou de papel;
- *state_at(Id, Tempo)* - retorna o estado que desempenhava a instância identificada por *Id* no instante de tempo *Tempo*;
- *lower_bound(Intervalo)* - retorna o limite inferior do intervalo *Intervalo*.

2.11. Especialização e Agregação

O modelo TF-ORM suporta mecanismos de especialização e agregação, semelhantes aos demais modelos orientados a objetos.

2.11.1. Declaração de Subclasse

Uma classe pode ser definida como subclasse de uma ou mais classes (herança múltipla). A declaração de uma subclasse é feita através de uma indicação feita ao lado de seu nome - acréscimo da cláusula *is_a* ao lado do nome da classe.

A definição de uma classe como subclasse de uma ou mais superclasses através do modelo TF-ORM apresenta as seguintes possibilidades de definição de papéis:

- **papéis herdados sem modificações** - todos os papéis das superclasses que não forem nomeados na definição da subclasse, com excessão dos papéis básicos, são herdados por esta subclasse; para efeito de clareza da especificação, pode ser utilizada a cláusula *inherits* identificando explicitamente estes papéis;
- **papéis estendidos** - um papel é estendido quando logo após a definição de seu nome for utilizada a cláusula *extends*, sendo listadas somente as novas

características deste papel; um papel com este nome deve existir em uma das superclasses; nos caso de herança múltipla, os nomes dos papéis podem ser associados ao nome da superclasse à qual pertencem;

- **papéis totalmente redefinidos** - quando um papel é definido com o nome igual a um outro papel existente em uma superclasse, sem a utilização da cláusula *extends*;
- **novos papéis definidos** - nos casos de definição de papéis com nomes diferentes de todos os papéis das superclasses; e
- **papéis não herdados** - esta possibilidade, que não está presente em nenhum dos modelos preliminares, permite que algum(s) comportamento(s) de uma superclasse não seja válido na subclasse; os papéis que não devem ser herdados são identificados através da cláusula *not-inherits;* nos casos de herança múltipla, cada papel listado nesta cláusula deve ser associado ao nome da superclasse à qual pertence.

O *papel básico* de uma subclasse deve ser uma união dos papéis básicos de todas as superclasses, considerando entretanto as alterações feitas nas definições dos papéis destas superclasses. Ao definir uma subclasse em TF-ORM: (i) as propriedades do papel básico da subclasse não necessitam ser definidas, sendo dadas pela união das propriedades dos papéis básicos de suas superclasses; e (ii) o conjunto de regras do papel básico da subclasse deve ser totalmente redefinido, evitando desta maneira problemas no controle dos papéis.

Um objeto instanciado em uma subclasse também é membro de cada uma de suas superclasses - fisicamente o mesmo objeto está representado em cada uma de suas superclasses. O identificador do objeto na subclasse e nas superclasses deve, portanto, ser o mesmo. O gerenciador de um banco de dados que implemente o modelo TF-ORM deverá, sempre que um objeto for instanciado em uma subclasse, verificar quais as suas superclasses para definir o identificador deste objeto nestas com o mesmo valor. No modelo, isto significa que os valores das propriedades *oId* da subclasse e das superclasses é o mesmo. Com os papéis o tratamento é semelhante. Considerando os papéis que forem herdados sem modificações e aqueles que forem somente estendidos, para cada papel instanciado em uma subclasse haverá em alguma das superclasses alguma instância deste papel com o mesmo identificador, ou seja, com o mesmo valor da propriedade *rId*. Os novos papéis definidos na subclasse, evidentemente, não apresentam instâncias de papéis iguais nas superclasses. Os papéis que forem redefinidos em uma subclasse são tratados como se fossem novos papéis, não havendo nenhuma ligação com o papel de mesmo nome da superclasse.

Conflitos de nomes de papéis herdados podem ocorrer em dois casos: (i) nos casos de herança múltipla quando duas superclasses apresentam papéis com nomes iguais; e (ii) nos casos em que nas superclasses houve redefinição de papéis de outras superclasses. O primeiro tipo de conflito deve ser resolvido no momento de definição da subclasse, permitindo somente a herança de um dos papéis conflitantes. Isto pode ser feito ou qualificando o nome dos papéis herdados através da indicação de qual a classe da qual o papel é herdado, ou desabilitando os papéis que não devem ser

considerados através da utilização da cláusula *not-inherits*, indicando também a que classe pertence cada um dos papéis desabilitados. Para o segundo caso de conflito de nomes o modelo TF-ORM considera como válido o nome mais próximo da hierarquia de herança (o papel redefinido na subclasse).

2.11.2. Declaração de Agregação

Uma classe pode ser declarada como sendo formada pela agregação de várias outras classes, através da cláusula *composed_of* colocada ao lado do nome da classe. A abstração de agregação não envolve herança. Entretanto, deve ser tomado algum cuidado com os nomes fornecidos para as propriedades e os papéis da nova classe - se forem utilizados nomes iguais aos presentes em alguma das classes componentes, estes estarão inacessíveis às instâncias da classe composta.

Um mesmo objeto pode ser componente de um ou mais objetos compostos.

Para que seja possível instanciar um objeto em uma classe composta é necessário que as instâncias de cada um de seus componentes estejam ativas.

3. Descrição da Aplicação

O caso descrito é bastante genérico e simples. Consideramos uma empresa composta por departamentos, um conjunto de empregados, gerentes e um diretor. Cada empregado é caracterizado por um nome, salário, sexo e data de nascimento. Um empregado trabalha em um departamento e pode apresentar uma ou mais habilitações para as quais está qualificado. O nome, salário, departamento e habilitações são propriedades que podem variar com o tempo, e por isso são declaradas como propriedades dinâmicas.

Os departamentos são caracterizados por um nome e por um gerente, que deve ser um dos empregados. Tanto o nome do departamento como o gerente deste podem variar com o tempo. Um departamento pode não apresentar um gerente por um período, por exemplo, em que uma substituição de gerentes está sendo efetuada, mas não pode nunca apresentar dois gerentes.

O gerente apresenta todas as características de um empregado, mas detém maior poder de decisão, podendo demitir e contratar empregados, aceitar e rejeitar pedidos de férias, modificar salários e habilitações. Só pode, porém, tomar decisões com relação aos empregados pertencentes ao seu departamento.

O diretor é um empregado que está acima dos gerentes e é capaz de criar, eliminar e renomear departamentos. Ele é capaz também de demitir empregados, promovê-los a gerentes (desde que existam departamentos sem gerente), aceitar ou rejeitar os pedidos de férias dos gerentes, alterar habilitações e salários.

Para modelar esta aplicação através do modelo TF-ORM, inicialmente foram escolhidas as classes que seriam definidas. Foi definida uma classe agente - *PESSOA*, uma classe recurso - *DEPARTAMENTO* e uma classe processo - *CONTROLE*. A classe *PESSOA* apresenta três papéis possíveis de serem desempenhados pelas suas instâncias: *Empregado, Gerente* e *Diretor*. A classe processo é uma classe que controla a passagem de mensagens entre os papéis da classe *PESSOA*, sendo dividida em três papéis: *Controle_do_empregado, Controle_do_gerente* e *Controle_do_diretor*.

Na figura 3.1 é representada graficamente a classe *PESSOA* com seus papéis, e as propriedades que armazenam as informações necessárias em cada um destes papéis. As propriedades comuns a todos os papéis são definidas no papel básico.

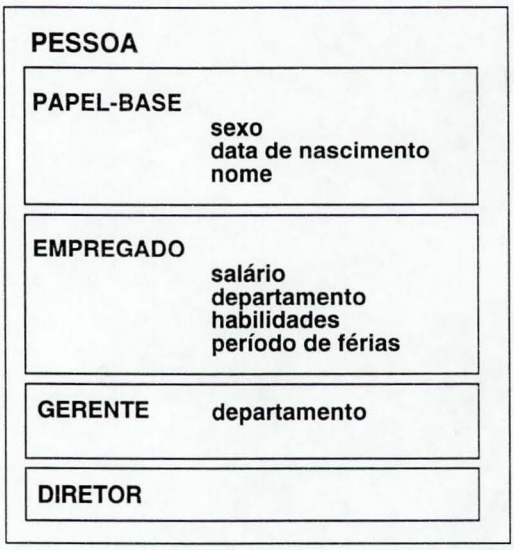

Figura 3.1: Lista de propriedades da classe pessoa

Uma vez definidas as classes e suas propriedades, passa-se à análise da comunicação entre as classes, representada no modelo através de mensagens. As possibilidades de interação entre os papéis da classe *PESSOA* e a classe recurso *DEPARTAMENTO* são representadas na figura 3.2.

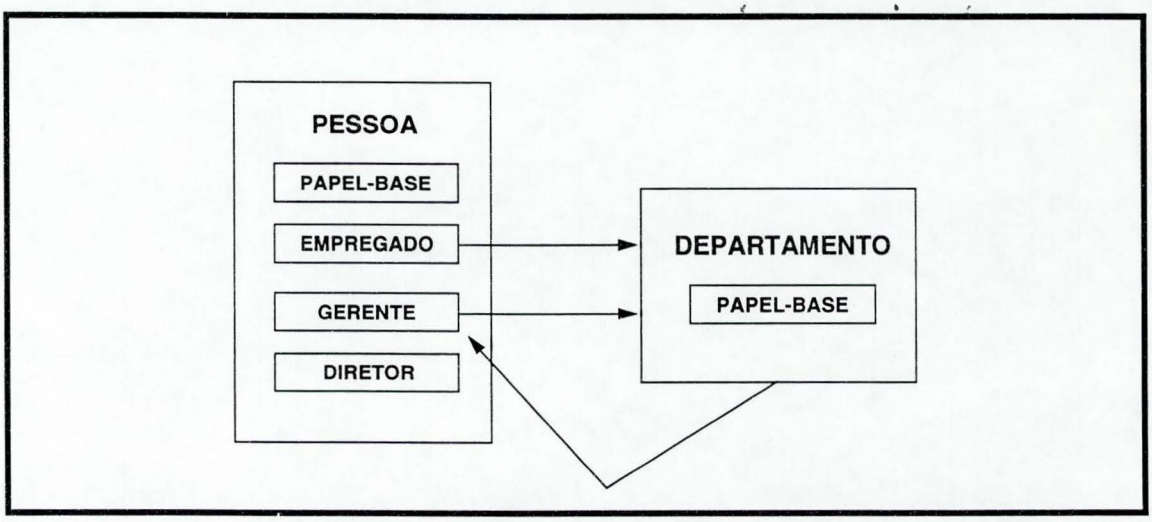

Figura 3.2: Modelagem das classes agente e recurso

4. Modelagem da Aplicação

```
agent class (
 PERSON,

 < Base_role,
 static properties = { (gender, {F,M}), (d_birth, date) },
 dynamic properties = { (name, string) },
 rules = {
    r1: state(active), msg(create_object) ⇒ msg(allow_role(employee)),
    r2: state(active), msg(create_object) ⇒ msg(allow_role(manager)),
    r3: state(active), msg(create_object) ⇒ msg(allow_role(director)) }
 >,

 < Employee,
 dynamic properties = {
    (cod, string),
    (salary, real),
    (work_dept, DEPARTMENT),
    (skill, set of string),
    (vacation, interval (closed, date) },
 states = { employed, not_employed, on_vacation, active, disabled },
 messages = {
    initial_values (Cod:string, Gender:{F,M}, DBirth:date, Name:string, Salary:real,
       Work_dept: string, Skill:set of string, Vacation:interval(closed,date))
       from CONTROL.employee_control,
    modify_name (Cod: string, NewName:string) to CONTROL.employee_control,
    modify_salary (Cod: string, Salary:real)from CONTROL.employee_control,
    add_skill (Cod: string, NewSkill:string) from CONTROL.employee_control,
    remove_skill (Cod: string, SkillName:string) from CONTROL.employee_control,
    change_dept (Cod: string, NewDept:department)
       from {CONTROL.employee_control, CONTROL.manager_control},
    ask_vacation (Cod: string, Begin:date, End:date) to CONTROL.manager_control,
    vacation_accepted (Cod: string, Begin: date, End: date)
       from CONTROL.manager_control,
    vacation_refused(Cod: string) from CONTROL.manager_control,
    begin_vacation (Cod: string) from CONTROL.manager_control,
    end_vacation (Cod: string) from CONTROL.manager_control,
    ask_dismissal (Cod: string) to CONTROL.manager_control,
    dismissal_accepted (Cod: string) from CONTROL.manager_control,
    dismissal (Cod: string) from CONTROL.employee_control },
 decisions = {
    new_name (Name: string),
    vacation_request (Begin:date, End:date),
    dismissal_request },
```

```
rules = {
   begin:
      msg(add_role) ⇒ state(active),
   initialization:
      state(active), msg(initial_values(Cod, Gender, DBirth, Name, Salary,
      Work_dept,   Skill, Vacation)) ⇒ state(employed);
      (not exists Rid (value(Rid, cod)=Cod)),
   changing name:
      state(employed), decision(new_name(Name)) ⇒
      msg(modify_name(Cod, Name)), state(employed),
   modifying_salary:
      state(employed), msg(modify_salary(Cod, NewValue)) ⇒
      state(employed); (value(salary)<NewValue) and (value(cod) = Cod),
   adding_skill:
      state(employed), msg(add_skill(Cod, NewSkill)) ⇒
      state(employed); (value(cod) = Cod),
   removing_skill:
      state(employed), msg(remove_skill(Cod, SkillName)) ⇒
      state(employed); (value(cod) = Cod),
   changing_department:
      state(employed), msg(change_dept(Cod, NewDept) ⇒
      state(employed); (value(cod) = Cod) and value(work_dept)<>NewDept),
   asking_vacation:
      state(employed), decision(vacation_request(Begin, End)) ⇒
      msg(ask_vacation(cod, Begin, End)), state(employed);
      (not exists Rid (has_role_instance(oId, manager, Rid)) and
      not exists Rid' (has_role_instance(oId, director, Rid')) and
      /* testa limites dos intervalos e se o período já não iniciou */
      ((Begin before End) and (Begin after now)),
   vacation_accepted_answer:
      state(employed), msg(vacation_accepted(Cod, Begin, End) ⇒
      state(employed); (value(cod) = Cod),
   vacation_refused_answer:
      state(employed), msg(vacation_refused(Cod)) ⇒
      state(employed); (value(cod) = Cod),
   beginning_vacation:
      state(employed), msg(begin_vacation(Cod)) ⇒
      state(on_vacation); (value(cod) = Cod),
   ending_vacation:
      state(on_vacation), msg(end_vacation(Cod)) ⇒
      state(employed); (value(cod) = Cod),
```

```
    requesting_dismissal:
        state(employed), decision(dismissal_request(Cod) ⇒
        msg(ask_dismissal(cod)), state(employed);
        (not exists Rid (has_role_instance(oId, manager, Rid)) and
        not exists Rid' (has_role_instance(oId, director, Rid'))),
    dismissal_accepted_answer:
        state(employed), msg(dismissal_accepted(Cod)) ⇒
        state(not_employed); (value(cod) = Cod),
    being_dismissed:
        state(employed), msg(dismissal(Cod)) ⇒ state(not_employed);
        (value(cod) = Cod) }
>,

< Manager,
dynamic properties = { (work_dept, DEPARTMENT) },
states = { active, disabled, on_vacation },
messages = {
    modify_employee_salary (Cod: string,Salary:real)
        to CONTROL.employee_control,
    add_employee_skill (Cod: string,NewSkill:string)
        to CONTROL.employee_control,
    remove_employee_skill (Cod: string, SkillName:string)
        to CONTROL.employee_control,
    change_employee_dept (Cod: string, NewDept:department)
        to CONTROL.employee_control,
    ask_employee_vacation (Cod: string, Begin:date, End:date)
        from CONTROL.manager_control,
    employee_vacation_accepted (Cod: string, Begin:date, End:date)
        to CONTROL.manager_control,
    employee_vacation_refused (Cod: string) to CONTROL.manager_control,
    employee_dismissal (Cod: string) to CONTROL.employee_control,
    add_manager_skill (Dept: DEPARTMENT, NewSkill:string)
        from CONTROL.manager_control,
    remove_manager_skill (Dept: DEPARTMENT , SkillName:string)
        from CONTROL.manager_control,
    change_manager_dept (Cod: string, Dept: DEPARTMENT, NewDept:department)
        from CONTROL.manager_control,
    ask_manager_vacation (Dept: DEPARTMENT, Begin:date, End:date)
        to CONTROL.director_control,
    vacation_manager_accepted (Dept: DEPARTMENT , Begin:date, End:date)
        from CONTROL.manager_control,
    vacation_manager_refused (Dept: DEPARTMENT )
        from CONTROL.manager_control,
    begin_manager_vacation (Cod: string, Dept: DEPARTMENT)
        from CONTROL.manager_control,
```

```
    end_manager_vacation (Cod: string, Dept: DEPARTMENT)
      from CONTROL.manager_control,
    rebaixar_manager (Dept: DEPARTMENT) from CONTROL.manager_control,
    ask_manager_dismissal (Cod: string, Dept: DEPARTMENT)
      to CONTROL.director_control,
    manager_dismissal_accepted (Dept: DEPARTMENT)
      from CONTROL.director_control,
    manager_dismissal (Dept: DEPARTMENT) from CONTROL.manager_control,
    add_employee (Cod: string, Gender:{F,M}, DBirth:date, Name:string,
      Salary:real, Work_dept: string, Skill:set of string,
      Vacation:interval(closed,date)) to CONTROL.employee_control  },
decisions = {
    modify_a_employee_salary (Cod: string, Salary:real),
    add_a_employee_skill (Cod: string, NewSkill:string),
    remove_a_employee_skill (Cod: string, SkillName:string),
    change_a_employee_dept (Cod: string, NewDept:DEPARTMENT),
    employee_vacation_accept (Cod: string, Begin:date, End:date),
    employee_vacation_refuse (Cod: string),
    a_employee_dismissal (Cod: string),
    ask_vacation_manager (Dept: DEPARTMENT, Cod: string, Begin:date,
      End:date),
    ask_dismissal_manager (Dept: DEPARTMENT, Cod: string),
    employee_add (Cod: string, Gender:{F,M}, DBirth:date, Name:string,
      Salary:real, Work_dept: string, Skill: set of string,
      Vacation:interval(closed,date))  }
rules = {
    begin:
      msg(add_role) ⇒ state(active);
      (exists Rid (has_role_instance (oId, employee, Rid)) and
      not exists Rid' (has_role_instance(oId, manager, Rid'))),
    modifying_employee_salary:
      state(active), decision (modify_a_employee_salary (Cod, Salary)) ⇒
      msg(modify_employee_salary (Cod, Salary)), state(active);
      (exists Rid (has_role_instance(Oid, employee, Rid) and
      value(Rid, cod)=Cod and value(Rid, word_dept)= value(work_dept)))
      /* empregado e gerente do mesmo dept.*/
    adding_employee_skill:
      state(active), decision(add_a_employee_skill(Cod, NewSkill)) ⇒
      msg(add_employee_skill(Cod, NewSkill)), state(active);
      (exists Rid (has_role_instance(Oid, employee, Rid) and
      value(Rid, cod)=Cod and value(Rid, word_dept)= value(work_dept))),
      /* empregado e gerente do mesmo dept.*/
```

removing_employee_skill:
state(active), decision(remove_a_employee_skill(Cod, SkillName)) ⇒
msg(remove_employee_skill(Cod, SkillName)), state(active);
(exists Rid (has_role_instance(Oid, employee, Rid) and
value(Rid, cod)=Cod and value(Rid, word_dept)= value(work_dept)) and
belongs(SkillName, value(Rid, skill))),
changing_employee_department:
state(active), decision(change_a_employee_dept(Cod, NewDept)) ⇒
msg(change_employee_dept(Cod, NewDept)), state(active);
(exists Rid (has_role_instance(Oid, employee, Rid) and
value(Rid, cod)=Cod and value(Rid, word_dept)= value(work_dept))),
/* empregado e gerente do mesmo dept.*/
employee_vacation_asked:
state(active), msg(ask_employee_vacation(Cod, Begin, End)) ⇒ state(active),
accepting_employee_vacation:
state(active), decision(employee_vacation_accepted(Cod, Begin, End)) ⇒
msg(employee_vacation_accepted(Cod, Begin, End)), state(active);
(exists Rid (has_role_instance(Oid, employee, Rid) and
value(Rid, cod)=Cod and value(Rid, word_dept)= value(work_dept))),
/* empregado e gerente do mesmo dept.*/
(Begin before End) and (Begin after now),
/* testa limites dos intervalos e se o período já não iniciou */
refusing_employee_vacation:
state(active), decision(employee_vacation_refuse(Cod)) ⇒
msg(employee_vacation_refused(Cod)), state(active);
(exists Rid (has_role_instance(Oid, employee, Rid) and
value(Rid, cod)=Cod and value(Rid, word_dept)= value(work_dept))),
/* empregado e gerente do mesmo dept.*/
dismissing_employee:
state(active), decision(a_employee_dismissal(Cod)) ⇒
msg(employee_dismissal(Cod)), state(active);
((not exists Rid (has_role_instance (OId, manager, Rid)) and
not exists Rid' (has_role_instance(OId, director, Rid')) and
exists Rid (has_role_instance(Oid, employee, Rid) and
value(Rid, cod)=Cod and value(Rid, word_dept)= value(work_dept))),
/* empregado e gerente do mesmo dept.*/
adding_manager_skill:
state(active), msg(add_manager_skill(Dept, NewSkill)) ⇒
state(active); (value(work_dept)=Dept),
removing_manager_skill:
state(active), msg(remove_manager_skill(Dept, SkillName)) ⇒
state(active); (value(work_dept)=Dept),
changing_manager_department:
state(active), msg(change_manager_dept(Cod, Dept, NewDept) ⇒
state(active) ; (value(work_dept)=Dept),

```
    manager_asking_vacation:
       state(active), decision(ask_vacation_manager(Dept,Cod, Begin, End)) ⇒
       msg(ask_manager_vacation(Dept , Begin, End), state(active);
       /* testa limites dos intervalos e se o período já não iniciou */
       (((Begin before End) and (Begin after now)) and (value(work_dept)=Dept)),
    manager_vacation_accepted:
       state(active), msg(vacation_manager_accepted(Dept, Begin, End)) ⇒
       state(active) ; (value(work_dept)=Dept),
    manager_vacation_refused:
       state(active), msg(vacation_manager_refused(Dept)) ⇒
       state(active) ; (value(work_dept)=Dept),
    beginning_manager_vacation:
       state(active), msg(begin_manager_vacation(Cod, Dept)) ⇒
       state(on_vacation) ; (value(work_dept)=Dept),
    ending_manager_vacation:
       state(on_vacation), msg(end_manager_vacation(Cod, Dept )) ⇒
       state(active) ; (value(work_dept)=Dept),
    rebaixando_manager:
       state(active), msg(rebaixar_manager(Dept)) ⇒
       state(disabled) ; (value(work_dept)=Dept),
    manager_asking_dismissal:
       state(active), decision(ask_dismissal_manager(Dept , Cod)) ⇒
       msg(ask_manager_dismissal(Dept, Cod)), state(active);
       (value(work_dept)=Dept),
    manager_dismissal_accepted_answer:
       state(active), msg(manager_dismissal_accepted(Dept)) ⇒
       state(disabled); (value(work_dept)=Dept),
    dismissing_manager:
       state(active), msg(manager_dismissal(Dept)) ⇒
       state(disabled); (value(work_dept)=Dept),
    adding_employee:
       state(active), decision(employee_add (Cod, Gender, DBirth, Name, Salary,
       Work_dept, Skill, Vacation)) ⇒ msg(add_employee (Cod, Gender, DBirth,
       Name, Salary, Work_dept, Skill, Vacation)), state(active)  }
>,

< Director,
states = { active, disabled },
messages = {
   promote_employee (Cod: string, ValidDate:date) to
CONTROL.employee_control,
   manager_rebaixar (Dept: DEPARTMENT) to CONTROL.manager_control,
   change_dept_manager(Cod: string, Dept: DEPARTMENT,
      NewDept:DEPARTMENT) to CONTROL.manager_control,
```

```
    modify_employee_salary (Cod: string, Salary:real)
      to CONTROL.employee_control,
    add_skill_manager (Dept: DEPARTMENT, NewSkill:string)
      to CONTROL.manager_control,
    remove_skill_manager (Dept: DEPARTMENT, SkillName:string)
      to CONTROL.manager_control,
    vacation_ask_manager (Dept: DEPARTMENT, Begin:date, End:date)
      from CONTROL.director_control,
    vacation_accepted_manager (Dept: DEPARTMENT, Begin:date, End:date)
      to CONTROL.manager_control,
    vacation_refused_manager (Dept: DEPARTMENT)
      to CONTROL.manager_control,
    dismissal_manager(Cod: string, Dept: DEPARTMENT)
      to CONTROL.manager_control,
    create_department (Name:string) to CONTROL.department_control,
    remove_department (Dept: DEPARTMENT) to CONTROL.department_control,
    rename_department (Dept: DEPARTMENT, NewName:string)
      to CONTROL.department_control,
    rebaixar_director (Cod: string) from CONTROL.director_control,
    ask_director_dismissal (Cod: string) to CONTROL.director_control,
    director_dismissal_accepted (Cod: string) from CONTROL.director_ control,
    director_dismissal (Cod: string) from CONTROL.director_control,
    add_employee (Cod: string, Gender:{F,M}, DBirth:date, Name:string,
      Salary:real, Work_dept: string, Skill:set of string,
      Vacation:interval(closed,date)) to CONTROL.employee_control  },
decisions = {
    employee_promote (Cod: string, ValidDate:date),
    a_manager_rebaixar (Dept: DEPARTMENT),
    manager_change_dept (Cod: string, Dept:DEPARTMENT,
          NewDept:DEPARTMENT),
    manager_modify_salary (Cod: string, Salary:real),
    manager_add_skill (Dept: DEPARTMENT, NewSkill:string),
    manager_remove_skill (Dept: DEPARTMENT, Skill:string),
    manager_vacation_accept (Dept: DEPARTMENT, Begin:date, End:date),
    manager_vacation_refuse (Dept: DEPARTMENT),
    a_manager_dismissal(Cod: string),
    department_create (Name:string),
    department_remove (Dept:DEPARTMENT),
    change_department_name (Dept:DEPARTMENT, NewName:string),
    ask_dismissal_director (Cod: string),
    add_a_employee((Cod: string, Gender:{F,M}, dBirth:date, Name:string,
      Salary:real, work_dept:string, skill:set of string,
      vacation:interval(closed,date))},
```

```
rules = {
   begin:
      msg(add_role) ⇒ state(active);
      (exists Rid (has_role_instance(oId, employee, Rid)) and
      not exists Rid'(has_role_instance(oId, manager, Rid')) and
      not exists Rid''(has_role_instance(OId, director, Rid''))),
      /* O novo diretor deve ser empregado, não podendo ser gerente e nem haver
      outro empregado que é diretor. */
   promoting_employee:
      state(active), decision(employee_promote(Cod, ValidDate)) ⇒
      msg(promote_employee(Cod, ValidDate)), state(active),
   rebaixando_manager:
      state(active), decision(a_manager_rebaixar(Dept, ValidDate)) ⇒
      msg(manager_rebaixar(Dept)), state(active),
   changing_manager_department:
      state(active), decision(manager_change_dept(Cod, Dept, NewDept)) ⇒
      msg(change_dept_manager(Cod, Dept, NewDept)), state(active),
   modifying_manager_salary:
      state(active), decision(manager_modify_salary(Cod, Salary)) ⇒
      msg(modify_employee_salary(Cod, Salary)), state(active),
   adding_manager_skill:
      state(active), decision(manager_add_skill(Dept, NewSkill)) ⇒
      msg(add_skill_manager(Dept, NewSkill)), state(active),
   removing_manager_skill:
      state(active), decision(manager_remove_skill(Dept, Skill)) ⇒
      msg(remove_skill_manager(Dept, Skill)), state(active);
      (belongs( SkillName, skill)),   /* só remove skill se estiver no conjunto */
   manager_vacation_request:
      state(active), msg(vacation_ask_manager(Dept, Begin, End)),
      decision(manager_vacation_accept(Dept, Begin, End)) ⇒
      msg(vacation_accepted_manager(Dept, Begin, End)), state(active);
      /* testa limites dos intervalos e se o período já não iniciou */
      ((Begin before End) and (Begin after now)),
   refusing_manager_vacation:
      state(active), msg(vacation_ask_manager(Dept, Begin, End)),
      decision(manager_vacation_refuse(Dept)) ⇒
      msg(vacation_refused_manager(Dept)), state(active),
   dismissing_manager:
      state(active), decision(a_manager_dismissal(Dept)) ⇒
      msg(dismissal_manager(Dept)), state(active),
   creating_department:
      state(active), decision(department_create(Name)) ⇒
      msg(create_department(Name)), state(active),
```

```
  removing_department:
      state(active), decision(department_remove(Dept)) ⇒
      msg(remove_department(Dept)), state(active);
      (not exists Rid (has_role_instance(Oid, employee, Rid) and
      Rid(value(Rid,work_dept)=Dept), /*não há nenhum emp. no departamento*/
    chnaging_department_name:
      state(active), decision(change_department_name(Dept, Name)) ⇒
      msg(rename_department(Dept, NewName)), state(active),
    rebaixando_director:
      state(active), msg(rebaixar_director(Cod)) ⇒ state(disabled),
    director_dismissal_request:
      state(active), decision(ask_dismissal_director(Cod)) ⇒
      msg(ask_director_dismissal(Cod)), state(active),
    director_dismissal_accepted_answer:
      state(active), msg(director_dismissal_accepted(Cod)) ⇒ state(disabled),
    dismissing_director:
      state(active), msg(director_dismissal(Cod) ⇒ state(disabled),
    adding_employee:
      state(active), decision(add_a_employee((Cod, Gender, DBirth, Name,
      Salary, Work_dept, Skill, Vacation)) ⇒
       msg(add_employee (Cod, Gender, DBirth, Name, Salary, Work_dept, Skill,
      Vacation)), state(active)  }
> )
```

```
resource class (
 DEPARTMENT,

 < Base_role,
 rules = {
    r1: msg(create_object) ⇒ state(active),
    r2: state(active) ⇒ add_role(Dept), state(active)  }
 >,

 < Dept,
 dynamic properties = { (name,string), (manager,PERSON) },
 messages = {
    new_department (DeptName:string) from CONTROL.department_control,
    exclude_department (DeptName: string) from CONTROL.department_control,
    department_rename (Dept:DEPARTMENT, NewName:string)
       from CONTROL.department_control,
    remove_manager (Dept:DEPARTMENT)
       from {CONTROL.manager_control, CONTROL.director_control },
    add_manager (Dept:DEPARTMENT, Cod: string)
       from CONTROL.employee_control  },
```

```
states = {disabled, active},
rules = {
   begin:
      state(disabled), msg(add_role) ⇒ state(active),
   initialization:
      msg(new_department(DeptName)) ⇒state(active);
      (not exists Rid(name=NewName)),
   excluding_department:
      state(active), msg(exclude_department(DeptName)) ⇒ state(disabled);
      (exists(Rid (Value(Rid,name) = DeptName)));
   renaming_department:
      state(active), msg(department_rename(Dept, NewName)) ⇒
      state(active); (not exists Rid (value(Rid,name) = NewName)),
   removing_manager:
      state(active), msg(remove_manager(Dept)) ⇒ state(active),
   adding_manager:
      state(active), msg(add_manager(Dept, Cod)) ⇒ state(active);
      ((exists Rid (has_role_instance(OId, employee, Rid)) and
      not exists Rid' (has_role_instance(OId, manager, Rid'))),
   manager_must_be_an_employee:
      constraint(manager <> null ⇒ exists Oid (has_class_instance(person, Oid))
      and exists Rid (has_role_instance(Oid, employee, Rid) and
      value(Rid, name) = manager)
      and exists Rid' (has_role_instance(Oid, manager, Rid') and
      value(Rid',dept)=value(oId)))  }
> )


process class (
 CONTROL,

 < Base_role,
 rules = {
    r1: msg(create_object) ⇒ state(active),
    r2: state(active) ⇒ msg(add_role(employee_control)), state(active),
    r3: state(active), msg(add_role(manager_control)) ⇒ state(active),
    r4: state(active), msg(add_role(director_control)) ⇒ state(active)  }
 >,

 < Employee_control,
 states = { active }
 messages = {
    initial_values (Cod: string, Gender:{F,M}, DBirth:date, Name:string, Salary:real,
       Work_dept:string, Skill:set of string, Vacation:interval(closed,date))
       to PERSON.employee,
```

```
    modify_name (Cod: string, NewName:string) from PERSON.employee,
    modify_salary (Cod: string, Salary:real) to PERSON.employee,
    add_skill (Cod: string, NewSkill:string) to PERSON.employee,
    remove_skill (Cod: string, SkillName:string) to PERSON.employee,
    change_dept (Cod: string, NewDept:department) to PERSON.employee,
    dismissal (Cod: string) to PERSON.employee,
    modify_employee_salary (Cod: string,Salary:real) from PERSON.manager,
    add_employee_skill (Cod: string,NewSkill:string) from PERSON.manager,
    remove_employee_skill (Cod: string, SkillName:string) from PERSON.manager,
    change_employee_dept (Cod: string, NewDept:department)
      from PERSON.manager,
    employee_dismissal (Cod: string)
      from  {PERSON.manager, CONTROL.manager_control,
      CONTROL.director_control},
    promote_employee (Cod: string)
      from {PERSON.director, CONTROL.manager_control},
    add_employee (Cod: string, Gender:{F,M}, DBirth: date, Name: string,
      Salary: real, Work_dept: string, Skill: set of string,
      Vacation: interval(closed,date))
      from {external world, PERSON.director, PERSON.manager},
    add_manager (Dept:DEPARTMENT, Cod: string) to DEPARTMENT.dept  },
rules = {
    begin:
      msg (add_role) ⇒ state(active),
    initialization:
      state(active), msg(add_employee(Cod, Gender, DBirth, Name, Salary,
      Work_dept, Skill, Vacation)) ⇒ msg(add_role(Person.Employee)),
      msg(initial_values(Cod, Gender, DBirth, Name, Salary, Work_dept, Skill,
      Vacation)), state(active),
    modifying_name:
      state(active), msg(modify_name(Cod, NewName)) ⇒ state(active),
    modifying_employee_salary:
      state(active), msg(modify_employee_salary(Cod, Salary) ⇒
      msg(modify_salary(Cod, Salary),state(active);
      (not exists Rid (has_role_instance(OId, manager, Rid) and
      not exists  Rid (value(Rid,cod) = Cod)),
    adding_employee_skill:
      state(active), msg(add_employee_skill(Cod, NewSkill)) ⇒
      msg(add_skill(Cod, NewSkill)), state(active),
    removing_employee_skill:
      state(active), msg(remove_employee_skill(Cod, SkillName)) ⇒
      msg(remove_skill(Cod, SkillName)), state(active),
```

changing_employee_department:
state(active), msg(change_employee_dept(Cod, NewDept)) ⇒
msg(change_dept(Cod, NewDept)), state(active),
(not exists Rid (has_role_instance (oId, manager, Rid)) and
not exists Rid' (has_role_instance(OId, director, Rid'))),
dismissing_employee:
state(active), msg(employee_dismissal(Cod)) ⇒
msg(dismissal(Cod)), state(disabled),
promoting_employee:
state(active), msg(promote_employee(Cod)) ⇒
msg(add_role(PERSON.manager, Rid)), msg(add_manager(Dept, Cod)),
state(active), }
>,

< **Manager_control**,
states = { active },
messages = {
ask_vacation (Cod: string, Begin:date, End:date) from PERSON.employee,
vacation_accepted (Cod, Bedin, End) to PERSON.employee,
vacation_refused (Cod) to PERSON.employee,
begin_vacation (Cod: string) to PERSON.employee,
end_vacation (Cod: string) to PERSON.employee,
ask_dismissal (Cod: string) from PERSON.employee,
dismissal_accepted (Cod: string) to PERSON.employee,
ask_employee_vacation (Cod: string, Begin:date, End:date) to PERSON.manager,
employee_vacation_accepted (Cod: string, Begin:date, End:date)
from PERSON.manager,
employee_vacation_refused (Cod: string) from PERSON.manager,
add_manager_skill (Dept: DEPARTMENT, NewSkill:string)
to PERSON.manager,
remove_manager_skill (Dept: DEPARTMENT, SkillName:string)
to PERSON.manager,
change_manager_dept (Cod: string, Dept: DEPARTMENT,
NewDept: department) to PERSON.manager,
vacation_manager_accepted (Dept: DEPARTMENT, Begin:date, End:date)
to PERSON.manager,
vacation_manager_refused (Dept: DEPARTMENT) to PERSON.manager,
begin_manager_vacation (Cod: string, Dept: DEPARTMENT)
to PERSON.manager,
end_manager_vacation (Cod: string, Dept: DEPARTMENT)
to PERSON.manager,
rebaixar_manager (Dept: DEPARTMENT) to PERSON.manager,
manager_dismissal (Dept: DEPARTMENT) to PERSON.manager,
dismissal_manager(Cod: string, Dept: DEPARTMENT) from PERSON.director,
manager_rebaixar (Cod: string) from PERSON.director,

```
    change_dept_manager(Cod: string, Dept:DEPARTMENT,
      NewDept:DEPARTMENT) from PERSON.director,
    change_dept (Cod: string, NewDept:department) to PERSON.employee,
    promote_employee (Cod: string) to CONTROL.employee_control,
    add_skill_manager (Dept: DEPARTMENT, NewSkill:string)
      from PERSON.director,
    remove_skill_manager (Dept: DEPARTMENT, SkillName:string)
      from PERSON.director,
    vacation_accepted_manager (Dept: DEPARTMENT, Begin:date, End:date)
       from PERSON.director,
    vacation_refused_manager (Dept: DEPARTMENT) from PERSON.director,
    remove_manager (Dept:DEPARTMENT) to DEPARTMENT.dept  },
rules = {
    begin:
      msg (add_role) ⇒ state(active),
    asking_employee_vacation:
      state(active), msg(ask_vacation (Cod, Begin, End) ⇒
      state(active), msg(ask_employee_vacation (Cod, Begin, End),
    asking_dismissal:
      state(active), msg(ask_dismissal(Cod)) ⇒
      msg (dismissal_accepted(Cod)), state(active),
    employee_vacation_accepted_answer:
      state(active), msg(employee_vacation_accepted(Cod, Begin, End) ⇒
      msg(vacation_accepted(Cod, Begin, End), state(active),
    employee_vacation_refused_answer:
      state(active), msg(employee_vacation_refused(Cod)) ⇒
      msg(vacation_refused(Cod)), state(active),
    adding_manager_skill:
      state(active), msg(add_skill_manager(Dept, NewSkill)) ⇒
      msg(add_manager_skill(Dept, NewSkill)), state(active),
    removing_manager_skill:
      state(active), msg(remove_skill_manager(Dept, SkillName) ⇒
      msg(remove_manager_skill(Dept, SkillName)), state(active),
    changing_manager_department:
      state(active),msg(change_dept_manager(Dept, NewDept)) ⇒
      msg(rebaixar_manager(Cod)), msg(remove_manager(Dept),
      msg(change_dept (Cod, NewDept)),
      msg(change_manager_dept (Cod, Dept, NewDept)),
      msg(promote_employee(Cod)), state(active);
      (exists Rid (has_role_instance(Oid, manager, Rid)) and
       value(NewDept.manager) = null ),
      /* Deve ser gerente e não haver outro gerente no departamento */
    rebaixando_manager:
      state(active), msg(manager_rebaixar(Dept)) ⇒
      msg(rebaixar_manager(Dept)), msg(remove_manager(Dept), state(active),
```

```
    manager_vacation_accepted_answer:
        state(active), msg(vacation_accepted_manager(Dept, Begin, End)) ⇒
        msg(vacation_manager_accepted(Dept, Begin, End)),
    manager_vacation_refused_answer:
        state(active), msg(vacation_refused_manager(Dept)) ⇒
        msg(vacation_manager_refused(Dept)), state(active),
    dismissing_manager:
        state(active), msg(dismissal_manager(Cod, Dept)) ⇒
        msg (manager_dismissal(Dept)), msg(employee_dismissal (Cod)),
        msg(remove_manager(Dept), state(disabled),
    beginning_vacation:
        state(active) ⇒ msg(begin_vacation(Cod)), state(active);
        (exists Rid (has_role_instance (oId, employee, Rid)) and
        (value(Rid, lower_bound(vacation)) = now),
    ending_vacation:
        state(active) ⇒ msg(end_vacation(Cod)), state(active);
        (exists Rid (has_role_instance (oId, employee, Rid)) and
        (value(Rid, upper_bound(vacation)) = now),
    beginning_manager_vacation:
        state(active) ⇒ msg(begin_manager_vacation(Dept )),
        /* O empregado correspondente também entra em férias*/
        msg(begin_vacation(Cod)), state(active);
        (exists Rid (has_role_instance (oId, manager, Rid)) and
        (exists Rid (has_role_instance (oId, employee, Rid)) and
        (value(Rid, lower_bound(vacation)) = now),
    ending_manager_vacation:
        state(active) ⇒ msg(end_manager_vacation(Dept)),
        /* O empregado correspondente também volta a trabalhar */
        msg(end_vacation(Cod)), state(active);
        (exists Rid (has_role_instance (oId, manager, Rid)) and
        (exists Rid (has_role_instance (oId, employee, Rid)) and
        (value(Rid, upper_bound(vacation)) = now),
>,

< Director_control,
states = { active },
messages = {
    employee_dismissal (Cod: string) to CONTROL.employee_control,
    remove_manager (Dept:DEPARTMENT) to DEPARTMENT.dept  }
    ask_manager_vacation (Dept: DEPARTMENT, Begin:date, End:date)
        from PERSON.manager,
    ask_manager_dismissal (Cod: string, Dept: DEPARTMENT)
        from PERSON.manager,
    manager_dismissal_accepted (Dept: DEPARTMENT) to PERSON.manager,
```

```
    vacation_ask_manager (Dept: DEPARTMENT, Begin:date, End:date)
      to PERSON.director,
    rebaixar_director (Cod: string) to PERSON.director,
    ask_director_dismissal (Cod: string) from PERSON.director,
    director_dismissal_accepted (Cod: string) to PERSON.director,
    director_dismissal (Cod: string) to PERSON.director,
    dismissal_director (Cod: string) from external world,
    director_rebaixar(Cod: string) from external world   }
rules = {
    begin:
      msg (add_role) ⇒ state(active),
    manager_vacation_request:
      state(active), msg(ask_manager_vacation(Dept, Begin, End)) ⇒
      msg(vacation_ask_manager(Dept, Begin, Date)), state(active),
    manager_dismissal_request:
      state(active), msg(ask_manager_dismissal(Cod, Dept)) ⇒
      msg(manager_dismissal_accepted(Dept)), msg(employee_dismissal (Cod)),
      msg(remove_manager(Dept), state(active),
    rebaixando_director:
      state(active), msg(director_rebaixar(Cod)) ⇒
      msg(rebaixar_director(Cod)), state(disabled),
    director_dismissal_request:
      state(active), msg(ask_director_dismissal(Cod)) ⇒
      msg(director_dismissal_accepted(Cod)), msg(employee_dismissal (Cod)),
      state(disabled),
    dismissing_director:
      state(active), msg(dismissal_director(Cod)) ⇒
      msg(director_dismissal(Cod)), msg(employee_dismissal (Cod)), state(disabled)
}
>,


< Department_control,
states = { active },
messages = {
    create_department (Name:string) from PERSON.director,
    remove_department (Dept: DEPARTMENT) from PERSON.director,
    rename_department (Dept: DEPARTMENT, NewName:string)
      from PERSON.director,
    new_department (DeptName:string) to DEPARTMENT.dept,
    exclude_department (DeptName: string) to DEPARTMENT.dept
    department_rename (Dept:DEPARTMENT, NewName:string)
      to DEPARTMENT.dept  },
rules = {
    begin:
      msg (add_role) ⇒ state(active),
```

```
    initialization:
        state(active), msg(create_department(Name)) ⇒
        msg(add_role(DEPARTMENT.dept), msg(new_department(DeptName)),
        state(active),
    removing_department:
        state(active), msg(remove_department(Dept)) ⇒
        msg(exclude_department(Dept)), state(disabled),
    renaming_department:
        state(active), msg(rename_department(Dept, NewName) ⇒
        msg(department_rename(Dept, NewName)), state(active)  }
>)
```

Anexo

Sintaxe da Linguagem de Definição TF-ORM

A notação utilizada é uma BNF ("Backus Naur Form" ou "Backus Normal Form") simplificada, utilizada em gramáticas livres do contexto. Cada unidade sintática da linguagem é definida através de uma regra de produção. O lado esquerdo de cada regra apresenta somente uma metavariável, separada do lado direito por "::=". Do lado direito das regras aparecem metavariáveis e terminais, além de símbolos especiais que denotam regras alternativas, partes opcionais e partes repetidas. As metavariáveis são delimitadas pelos símbolos "<" e ">". Os terminais podem ser seqüências de caracteres ou símbolos. As seqüências de caracteres são representadas por elas mesmas, em negrito; os símbolos são delimitados por um par de duplas aspas. O símbolo especial "|" separa duas alternativas para a definição da mesma metavariável. Conjuntos de elementos que são opcionais são delimitados pelos símbolos "[" e "]". Conjuntos de elementos que podem ser repetidos zero ou mais vezes são delimitados pelos símbolos "{" e "}*". E conjuntos de símbolos que podem ser repretidos uma ou mais vezes são delimitados pelos símbolos "{" e "}$^{+}$". A precedência dos operadores é a seguinte: opcional, repetição, seqüência e alternativas.

As regras de produção correspondentes à linguagem de definição de dados do modelo TF-ORM é a seguinte:

<class declaration> ::=
  <process class declaration>
| <resouce class declaration>
| <agent class declaration>
<process class declaration> ::= **process class** <process class definition>
<resource class declaration> ::= **resource class** <resource class definition>
<agent class declaration> ::= **agent class** <agent class definition>
<process class definition> ::=
  "(" <class name> "," <base-role declaration>
    {"," <process role declaration> }* ")"
| "(" <class name> <specialization declaration> ","
    [<disabled roles declaration> ","]
    [<extended role declaration> ","]
      <base-role declaration> { "," <process role declaration>}* ")"
<resource class definition> ::=
  "(" <class name> "," <base-role declaration>
    { "," <resource role declaration> }* ")"
| "(" <class name> <specialization declaration> ","

[<inherited roles declarations> ","]
[<disabled roles declaration> ","]
[<extended role declaration> ","]
<base-role declaration> { "," <resource role declaration> }* ")"
| "(" <class name> <component declaration> ","
[<disabled roles declaration> ","]
< base-role declaration> { "," <resource role declaration> }* ")"

<agent class definition> ::=
"(" <class name> "," <base-role declaration>
{ "," <agent role declaration> }* ")"
| "(" <class name> <specialization declaration> ","
[<inherited roles declaration> ","]
[<disabled roles declaration> ","]
[<extended role declaration> ","]
<base-role declaration> { "," <agent role declaration> }* ")"

<class name> ::= <identifier>

<identifier> ::= <letter> { <identifier character> }*

<identifier character> ::= <letter> | <digit> | "_"

<specialization declaration> ::= **is_a** <class name>
| **is_a** "(" <class name> { "," <class name> }+ ")"

<inherited roles declaration> ::=
inherits [<class name> "."] <role name>
| **inherits** "(" [<class name> "."] <role name>
{ "," [<class name> "."] <role name> }* ")"

<disabled roles declaration> ::=
not_inherits [<class name> "."] <role name>
| **not_inherits** "(" [<class name> "."] <role name>
{ "," [<class name> "."] <role name> }* ")"

<extended role declaration> ::=
extends [<class name> "."] <role name>
| **extends** "(" [<class name> "."] <role name>
{ "," [<class name> "."] <role name> }* ")"

<component declaration> ::=
composed_of "{" <class name> { "," <class name> }* "}"

<role name> ::= <identifier>

<base-role declaration> ::= "<" **base_role** ["," <static properties declaration>]
["," <dynamic properties declaration>] "," <rule declaration> ">"

<process role declaration> ::= "<" <process role name>
["," <static properties declaration>]
["," <dynamic process properties declaration>]
< message declaration> "," <state declaration> "," <rule declaration> ">"

<resource role declaration> ::= "<" <resource role name>
["," <static properties declaration>]
["," <dynamic properties declaration>]

< message declaration> "," <state declaration> "," <rule declaration> ">"
<agent role declaration> ::= "<" <agent role name>
["," <static properties declaration>]
["," <dynamic properties declaration>]
["," <decision declaration>]
<message declaration> "," <state declaration> ","
<agent rule declaration> ">"
<process role name> ::= <agent name> "." <activity name> | <activity name>
<agent name> ::= [<class name> "."] <agent role name>
<activity name> ::= <identifier>
<resource role name> ::= <identifier>
<agent role name> ::= <identifier>
<static properties declaration> ::= **static properties** "=" "{" [<property list>] "}"
<dynamic process properties declaration> ::= **dynamic properties** "=" "{"
[**executing agent** "," <class name> ","]
[**message senders** "," "{" <sender list> "}" ","]
[**message receivers** "," "{" <sender list> "}" ","]
[<property list>] "}"
<sender list> ::= <sender> | <sender> { "," <sender> }
<sender> ::= <class name> | <class name> "." <role name> | <role name>
<dynamic properties declaration> ::= **dynamic properties** "=" "{"
<property list> "}"
<property list> ::= "(" <property name> "," <property domain> ")" ["," <property list>]
<property name> ::= <identifier>
<property domain> ::= <simple domain> | <complex domain>
<simple domain> ::= <class name> ["." <role name>]
| <role name>
| <predefined domain>
| "{" <string list> "}"
<predefined domain> ::= **integer** | **real** | **boolean** | **string** | **text** | **place** | **title** | **image**
| <temporal point type> | <interval> | | <limit>
<temporal point type> ::= **instant** | **date** | **time** | **year** | **month** | **day** | **hour** | **minute**
| **week** | **semester** | **century** | **weekday**
<interval> ::= **interval** "(" <interval type> "," <interval limits> ")"
<interval type> ::= **closed** | **open** | **open_down** | **open_up** | **floating_down** | **floating_up**
<interval limits> ::= **instant** | **date** | **time** | **year** | **month** | **day** | **hour** | **minute**
 ::= **span** "(" <span type> ")"
<span type> ::= **year** | **month** | **day** | **hour** | **minute**| **week** | **semester** | **century**
<limit> ::= **after** "(" <limit type> ")"
| **before** "(" <limit type> ")"
| **within** "(" <interval limits> ")"
<limit type> ::= **instant** | **date** | **time** | **year** | **month** | **day** | **hour** | **minute**
<string list> ::= <identifier> ["," <string list>]
<complex domain> ::= "{" <simple domain> "}" | "{" <property list> "}"

| "(" property list> ")" | **set_of** <simple domain> | **list_of** <simple domain>
<decision declaration> ::= **decisions** "=" "{"
<decision definition> { "," <decision definition> }* "}"
<decision definition> ::= <decision>
<decision> ::= <decision name> ["(" <decision parameters declaration> ")"]
<decision name> ::= <identifier>
<decision parameters declaration> ::=
<parameter declaration>{ "," <parameter declaration> }*
["," **valid_time** ":" <date value>]
<parameter declaration> ::= <parameter name> ":" <property domain>
| <property name>
<parameter name>::= <identifier>
<message declaration > ::= **messages** "=" "{"
<message definition> { "," <message definition> }* "}"
<message definition> ::= <message> **to** <roles>
| <message> to **EXTERNAL_WORLD**
| <message> **to itself**
| <message> **from** <roles>
| <message> from **EXTERNAL_WORLD**
<message> ::= <message name> ["(" <message parameters declaration> ")"]
<message name> ::= <identifier>
<message parameters declaration> ::=
<parameter declaration> { "," <parameter declaration> }*
["," **valid_time** ":" <date value>]
<roles> ::= <role> | "{" <role> { "," <role> }* "}"
<role> ::= [<class name> "."] <role name>
<state declaration> ::= **states** "=" "{" [<state> { "," <state> }*] "}"
<state> ::= <state name> | "(" <state name> "," <state name> { "," <state name> }*
")"
<state name> ::= <identifier>
<rule declaration> ::= **rules** "="
"{" [<rule name> ":" <rule> { "," <rule name> ":" <rule> }*] "}"
<rule name> ::= <identifier>
<rule> := <state transition rule> | <integrity rule>
<agent rule declaration> ::= **rules** "="
"{" [<rule name> ":" <agent rule> { "," <rule name> ":" <agent rule> }*]
"}"
<agent rule> := <agent state transition rule> | <integrity rule>
<state transition rule> ::=
<left predicate> "⇒" <right predicate> [<state transition condition>]
<left predicate> ::= <state predicate> "," [<message predicate in>]
| <message predicate in>
<agent state transition rule> ::=

<agent left predicate> "⇒" <right predicate> [<state transition condition>]

<agent left predicate> ::= <state predicate> ["," <message predicate in>]
| <message predicate in>
| <state predicate> ["," <decision predicate>]
| <decision predicate>

<right predicate> ::= <message predicate out> ["," <right predicate>]
| <state predicate>

<state predicate> ::= <defined state predicate> | <predefined state predicate>

<defined state predicate> ::= **state** "(" [<id variable> ","] <state name> ")"

<predefined state predicate> ::= **state** "(" [<oid variable> ","] <predefined state> ")"

<predefined state> ::= **active** | **suspended**

<message predicate in> ::= **msg** "(" <message in> ")"
| "{" <message in list> "}"

<message in list> ::= <message in> { "," <message in> }*

<message in> ::= <message name> ["(" <message parameters> ")"]
| <predefined message predicate>

<decision predicate> ::=
decision "(" <decision name> ["(" <message parameters> ")"] ")"

<message parameters> ::= <parameter> { "," <parameter> }*

<parameter> ::= <parameter name>

<predefined message predicate> ::=
create_object ["(" <class name> "," < oid variable> ")"]
| **suspend_object** ["(" <oid variable> ")"]
| **resume_object** ["(" <oid variable> ")"]
| **kill** ["(" <oid variable> ")"]
| **kill** "(" **itself** ")"
| **add_role** ["(" [<oid variable> ","] <role> [","<role variable>] ")"]
| **suspend_role** ["(" <role variable> ")"]
| **resume_role** ["(" <role variable> ")"]
| **terminate_role** ["(" <role variable> ")"]
| **forbid_role** ["(" [<role variable> ","] <role> ")"]
| **allow_role** ["(" [<oid variable> ","] <role> ")"]
| **forbid_op** "(" [<role variable> ","] <direction> <message name> ")"
| **allow_op** "(" <role variable> "," <direction> <message name> ")"
| **forget** "(" [<oid variable> ","] <rule name> ")"
| **recall** "(" [<oid variable> ","] <rule name> ")"
| **start** ["(" [<oid variable> ","] <role> ["," <role variable>] ")"]
| **stop** ["(" [<oid variable> ","] <role> ["," <role variable>] ")"]
| **in_class** "(" <class name> ")"
| **out_class** "(" <class name> ")"

<direction> ::= "←" | "→"

<message predicate out> ::= **msg** "(" <message out> ")"

<message out> ::= <message name> ["(" <message parameters> ")"]
| <predefined message predicate> [**to** <receiver>]

<receiver> ::= [<role variable> ":"] <role> | **itself**
<oid variable> ::= <variable>
<role variable> ::= <variable>
<id variable> ::= <oid variable> | <role variable>
<state transition condition> ::= ";" "(" <logical expression> ")"
<integrity rule> ::= **constraint** "(" <integrity condition declaration> ")"
<integrity condition declaration> ::=
    <simple integrity condition> | <instanciated integrity condition>
<simple integrity condition> ::= <logical expression> "⇒" <logical expression>
<instanciated integrity condition> ::= [<temporal operator>] <quantifier> <variable>
    "(" <integrity condition declaration> ")"
<logical expression> ::= <logical term>
    | <logical expression> <or operator> <logical term>
<logical term> ::= <logical factor>
    | <logical term> <and operator> <logical factor>
<logical factor> ::= <logical element> | **not** <logical element>
<logical element> ::= <predicate> | "(" <logical expression> ")"
    | <logical element> <temporal logical operator> <predicate>
<or operator> ::= **or** | ";"
<and operator> ::= **and** | ","
<temporal logical operator> ::= **since** | **until** | **before** | **after**
<predicate> ::= <predefined predicate>
    | <predefined temporal predicate>
    | <state predicate>
    | <predefined state predicate>
    | [<temporal operator>] <quantifier> <variable> "(" <predicate> ")"
    | <arit expression> <comp operator> <arit expression>
    | <temporal operator> <logical expression>
    | **false**
    | **true**
<predefined predicate> ::=
    has_class_instance "(" <class name> "," <oid variable> ")"
    | **has_role_instance** "(" <oid variable> "," <role name> "," <role variable> ")"
    | **active_class** "(" <oid variable> ")"
    | **active_role** "(" <role variable> ")"
    | **active_class_at** "(" <oid variable> ","<temporal instant> ")"
    | **active_role_at** "(" <role variable> "," <temporal instant> ")"
    | **is_valid** "(" <id variable> "," <property name> ")"
    | **is_valid_at** "(" <id variable> "," <property name> "," <temporal instant> ")"
    | **out_role** "(" <oid variable> "," role name> ")"
    | **role** "(" <oid variable> "," <role variable> ")"
<predefined temporal predicate> ::=
    belongs "(" <function argument> "," <function name argument> ")"
    | **contains** "(" <function name argument> "," <function argument> ")"
    | **before** "(" <function argument> "," <function argument> ")"
    | **equal** "(" <function argument> "," <function argument> ")"

<function argument> ::= [<id variable> ","] <property name>
| <temporal instant> | <variable>
<function name argument> ::=
[<id variable> ","] <property name> | <variable>
<temporal instant> ::= <number> "/" <number> "/" <number> ","
<number> ":" <number>
<temporal operator> ::= **sometime past** | **immediately past** | **always past**
sometime future | **immediately future** | **always future**
<quantifier> ::= **exists** | **forall**
<comp operator> ::= "<" | ">" | "=" | "≤" | "≥" | "≠"
<arit expression> ::= <term> | "-" <arit expression>
| <arit expression> "+" <term> | <arit expression> "-" <term>
<term> ::= <factor> | <term> "*" <factor> | <term> "/" <factor>
<factor> ::= <element> | <factor> "**" <element>
| <element> **union** <element>
| <element> **intersection** <element>
<element> ::=
[<id variable> ","] <property name>
| [<id varible> ","] <predefined property name>
| <function> | <value> | <variable> | "(" <arit expression> ")" | **now**
<predefined property name> ::= **oId** | **object_instance** | **end_object**
| **rId** | **role_instance** | **end_role**
<function> ::=
year "(" <function argument> ")"
| **month** "(" <function argument> ")"
| **day** "(" <function argument> ")"
| **hour** "(" <function argument> ")"
| **minute** "(" <function argument> ")"
| **weekday** "(" <function argument> ")"
| **lower_bound** "(" <function name argument> ")"
| **upper_bound** "(" <function name argument> ")"
| **duration** "(" <function name argument> ")"
| **interval** "("<function name argument>","<function name argument> ")"
| **to_minutes** "(" <function argument> ")"
| **to_months** "(" <function argument> ")"
| **to_days** "(" <function argument> ")"
| **value** "(" [<id variable> ","] <property name> ")"
| **past_value** "("[<id variable>","]<property name><temporal instant> ")"
| **valid_time** "(" [<id variable> ","] <property name> ")"
| **transaction_time** "(" [<id variable> ","] <property name> ")"
| **class_creation_time** "(" <oid variable> ")"
| **role_creation_time** "(" <role variable> ")"
| **class_end_time** "(" <oid variable> ")"
| **role_end_time** "(" <role variable> ")"
| **state** "(" <id variable> ")"

| **state_at** "(" <id variable> <temporal instant> ")"

<value> ::= <integer number> | <string> | <temporal value> | **null**

<integer number> ::= <digit> { <digit> }*

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<string> ::= " ' " { <any character including blanck> }$^{+}$ " ' "

<temporal value> ::= <temporal instant> | <date value> | <hour value>

<date value> ::= <number> "/" <number> "/" <number>

<hour value> ::= <number> ":" <number>

<number> ::= <digit> <digit>

<variable> ::= <identifier>

<identifier> ::= <letter> { <letter> | <digit> | "_" }*

<letter> ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N
| O | P | Q | R | S | T | U | V | W | X | Y | Z
| a | b | c | d | e | f | g | h | i | j | k | l | m | n
| o | p | q | r | s | t | u | v | w | x | y | z

Referências Bibliográficas

[ADI 87] ADIBA, M.; QUANG, N.B.; COLLET, C. Aspects Temporals, Historiques et Dynamiques des Bases de Donnés. *TSI - Technique et Science Informatiques*, AFCET-Bordas, v.6, n.5, p.457-478, 1987.

[CAS 88] CASAIS, E. *An Object Oriented System Implementating KNOs*. In: Conference of Office Information Systems, Palo Alto, California, 23-25 Mar. 1988. *Proceedings*. Ed. Robert B. Allen, 1988. p.284-90.

[CLI 88] CLIFFORD, J.; RAO, A. A Simple, general structure for temporal domains. In: C. Rolland; F. Bodart; M. Leonard (eds.) *Temporal Aspects in Information Systems*. Amsterdam, North-Holland, 1988. p.17-28.

[DEA 91] De ANTONELLIS, V.; PERNICI, B.; SAMARATI, P. F-ORM Method: a F-ORM Methodology for reusing specifications. In: F.V. Assche; B. Moulin; C. Rolland (eds.) *Object Oriented Approach in Information Systems*. Amsterdam: North-Holland, 1991. p.117-35.

[EDE 93a] EDELWEISS, N.; OLIVEIRA, J. P. M.; PERNICI, B. An Object-oriented temporal model. *Proceedings* of the 5th International Conference on Advanced Information Systems Engineering - CAISE'93, Paris, France, June 8-11, 1993. pp.397-415. (Lecture Notes in Computer Science n. 685).

[EDE 93b] EDELWEISS, N.; OLIVEIRA, J.P.M. CASTILHO, J.M.V. A Temporal Logic Language for Temporal Conditions Definition. *Proceedings of the 13th International Conference of the Chilean Computer Science Society*, La Serena, Chile, 14-16 outubro, 1993. pp.163-178.

[EDE 94] EDELWEISS, N. *Sistemas de Informação de Escritórios: um Modelo para Especificações Formais*. Porto Alegre: CPGCC/UFRGS, junho 1994. 187p. (Tese de doutorado).

[END 72] ENDERTON. *A Mathematical Introduction to Logic*. Addison-Wesley, 1972.

[JEN 94] JENSEN, C. S. (ed.). A Consensus glossary of temporal database concepts. *SIGMOD RECORD*, v.23, n.1, p.52-63, Mar. 1994.

[MAN 86] MANOLA, F.; DAYAL, U. PDM: An Object-Oriented Data Model. *Proceedings* of the International Workshop on Object Oriented Database Systems, Pacific Grove, California, 23-6 Sept. 1986.. p.18-25.

[PER 90] PERNICI, B. Objects with Roles. *SIGOIS Bulletin*, v.11, n.2-3, p.205-15, 1990.

[SNO 85] SNODGRASS, R.; AHN, I. A Taxonomy of time in databases. *Proceedings* of the ACM SIGMOD International Conference on Management of Data, Texas, May 28-31, 1985. p.236-46.

[TSI 87] TSICHRITZIS, D. et al. KNOs: Knowledge Acquisition, Dissemination, and Manipulation Objects. *ACM Transactions on Office Information Systems*, New York, v.5, n.1, p.96-112, Jan.1987.

[WIE 91] WIEDERHOLD, G.; JAJODIA, S.; LITWIN, W. Dealing with granularity of time in temporal databases. *Proceedings* of the 3rd International Conference on Advanced Information Systems Engineering - CAISE'91, 3., Trondheim, Norway, May 13-15, 1991. p.124-40.